White Paper

An Introduction to Fabasoft app.ducx

2013 Spring Release

Fabasoft®

Contents

1 Introduction **13**
1.1 Fabasoft's software product portfolio 13
1.1.1 Fabasoft app.ducx 13
1.1.2 Fabasoft Folio 13
1.1.3 Fabasoft Mindbreeze Enterprise 14
1.1.4 Fabasoft app.telemetry 14
1.2 Benefits of creating CCAs based on Fabasoft technology 14
1.2.1 Content governance 15
1.2.2 Single-instance content store 15
1.2.3 Compliance 15
1.2.4 Content life cycle management 15
1.2.5 Access protection and access rights system 15
1.2.6 Fabasoft reference architecture 15
1.2.7 Platform independence 15
1.3 Benefits of Fabasoft app.ducx 16
1.3.1 Shorter time to value 16
1.3.2 Holistic application life cycle management 16
1.3.3 Integrated in your development environment 17
1.3.4 Automated testing 17
1.3.5 Localization and customization 17
1.3.6 Integrated compiler 17
1.3.7 Delta loader 17
1.3.8 Automatic address management 17
1.3.9 Fabasoft app.ducx cache 17
1.3.10 "Convention over configuration" principle 17
1.4 Who should read this book? 18
1.5 General remarks concerning examples 18
2 Installing Fabasoft app.ducx **18**
2.1 Software requirements 18
2.2 Licenses 18
2.3 Which software do I need for using Fabasoft app.ducx? 19
2.4 Using Eclipse 19
2.4.1 Installing the Fabasoft app.ducx feature using the Update Site 19
2.4.2 Installing the Fabasoft app.ducx feature using the archive 20
2.4.3 Updating the Fabasoft app.ducx feature 21
2.4.4 Improving the performance of the Fabasoft app.ducx compiler 21

2.4.5 Using Fabasoft app.ducx with an older Fabasoft Folio Domain 21
3 General Use of Fabasoft app.ducx 21
3.1 General structure of a Fabasoft app.ducx project 22
3.2 Creating Fabasoft app.ducx projects using Eclipse 22
3.2.1 Fabasoft app.ducx-specific settings in the Eclipse preferences 22
3.2.2 Creating a new Fabasoft app.ducx project 23
3.2.3 Creating an Fabasoft app.ducx project from an existing software component 28
3.3 Working with Fabasoft app.ducx projects using Eclipse 29
3.3.1 Running a Fabasoft app.ducx project 29
3.3.2 Adding a software component reference 30
3.3.3 Adding a source file 31
3.3.4 Adding resources 32
3.3.5 Exporting a Fabasoft app.ducx project 32
3.3.6 Managing address ranges 33
3.3.7 Defining a friend component 35
3.3.8 Adding additional contents 35
3.3.9 Change references with refactoring 36
3.3.10 Change relation with refactoring 37
3.3.11 Working together/using a version control system 37
3.3.12 Customize font and colors 38
3.4 Updating Fabasoft app.ducx projects using Eclipse 38
3.5 Build and test environments 39
3.5.1 Prerequisites 39
3.5.2 Environment variables 39
3.5.3 Execute Fabasoft app.ducx Ant tasks 40
3.5.4 Fabasoft app.ducx Ant tasks 40
3.5.5 Adding dynamic contents to the software component 43
3.5.6 Update app.ducx libraries 43
3.6 Creating your own setup kit 43
3.7 Productivity features of Fabasoft app.ducx using Eclipse 44
3.7.1 Syntax Highlighting of Enumeration Values 44
3.7.2 Referenced Form Pages in Outline 45
3.7.3 Highlighting of the current domain specific language editor 45
3.7.4 Breadcrumb Navigation 46
3.7.5 Static check of generic assignments 46
3.7.6 Navigation between errors and warnings 48
3.7.7 Linking between Outline and Editor 48

3.7.8 Quick Outline 48
3.7.9 Find References 48
3.7.10 Mark Occurrences 49
3.7.11 Code Formatting 49
3.7.12 Color picker in the Content Assist of the UI Editor 50
3.7.13 Expression informationen in complex assignments 50
3.7.14 Information about variables in Expressions 50
3.7.15 Goto definition for variables in Expressions 50
3.7.16 Copy a reference from the Project Explorer 50
3.7.17 Outline in the Business Process Editor 50
3.7.18 Warning for missing user interface binding at object class declarations 51
3.7.19 Error for mismatching component and model 51
3.7.20 Quick fix to add a reference to the project 51
3.7.21 Content Assist 52
3.7.22 Quick Fix for ambigious elements 52
3.7.23 Folding 53
3.7.24 Automatically close brackets 53
3.7.25 Find the matching bracket 53
3.7.26 Tooltips 53
3.7.27 Edit component object instances utilizing the Fabasoft Folio Web Client 53
3.7.28 Templates 53
3.7.29 Spell checker 54
3.7.30 Fabasoft Reference Documentation 54
4 Domain-Specific Language Fundamentals 55
4.1 What is a DSL? 55
4.2 Common characteristics of DSLs 56
4.2.1 Keywords, expressions and blocks 56
4.2.2 Operators and punctuators 56
4.2.3 Comments 56
4.2.4 Multilingual strings 56
4.2.5 Using fully qualified references for referring to component objects 58
4.2.6 Using the import statement 58
4.2.7 Resolving of not fully qualified references 59
4.2.8 Resolving of qualifiers in app.ducx expressions 59
4.2.9 Using generic assignment statements 61
4.2.10 Public, private and obsolete 63
4.2.11 Referencing resources 64

4.3 Reference naming conventions 64
5 app.ducx Object Model Language 65
5.1 Defining an object class 66
5.1.1 Selecting the meta class 66
5.1.2 Defining the base class 66
5.1.3 Defining miscellaneous object class aspects 67
5.2 Adding properties to an object class 69
5.2.1 Reusing existing properties 69
5.2.2 Defining new properties 69
5.2.3 Property modifier prefixes 81
5.2.4 Property modifier suffixes 82
5.3 Extended property definitions 83
5.3.1 Initializing a property with a value 83
5.3.2 Protecting a property with access types 84
5.3.3 Assigning triggers to a property 84
5.3.4 Assigning constraints to a property 86
5.3.5 Object pointer property containing children 91
5.3.6 Object pointer property describing a hierarchy 91
5.4 Defining fields 92
5.5 Extending existing component objects 92
5.5.1 Extending an existing object class 92
5.5.2 Extending an existing enumeration type 93
5.5.3 Extending an existing compound type 94
5.6 Defining relations 94
5.7 Defining and extending component object instances 95
5.7.1 Defining a new component object instance 95
5.7.2 Extending an existing component object instance 96
5.8 Software products, software editions and software solutions 97
5.8.1 Software product 97
5.8.2 Software edition 97
5.8.3 Software solution 98
5.8.4 Example 98
6 app.ducx Resource Language 99
6.1 Defining strings 99
6.2 Defining error messages 99
6.3 Defining symbols 100
7 app.ducx User Interface Language 101

7.1 Defining forms and form pages 101
7.1.1 Defining a form 101
7.1.2 Defining a form page 103
7.1.3 Using the form designer 112
7.1.4 Defining a desk form 113
7.2 Extending existing desk forms, forms and form pages 114
7.2.1 Extending an existing desk form 114
7.2.2 Extending an existing form 114
7.2.3 Extending an existing form page 115
7.3 Defining menus, button bars and task panes 116
7.3.1 Defining a menu 116
7.3.2 Extending an existing menu 118
7.3.3 Weighted menu 118
7.3.4 Defining a task pane 119
7.3.5 Extending an existing task pane 120
7.4 Assigning user interface elements to an object class 120
7.4.1 Assigning a symbol to an object class 121
7.4.2 Assigning menus to an object class 121
7.4.3 Assigning menu items to an object class (Expansion Points) 123
7.4.4 Assigning task panes to an object class 125
7.4.5 Assigning forms to an object class 126
7.5 Defining portals 129
8 app.ducx Use Case Language 129
8.1 Declaring transaction variables 130
8.2 Defining use cases 130
8.2.1 Defining a new use case 131
8.2.2 Defining a new menu use case 135
8.3 Defining a virtual application 137
8.3.1 Implementing a virtual application 139
8.3.2 Extending a virtual application 144
8.3.3 Defining a dialog 144
8.3.4 Extending a dialog 151
8.4 Implementing a use case in Java 151
8.4.1 Defining a use case to be implemented in Java 153
8.4.2 Implementing the Java method 153
8.4.3 Importing packages generated by Fabasoft app.ducx 155
8.4.4 Data types in Java 155

8.4.5 Accessing properties and invoking use cases 156
8.4.6 Working with use case parameters 157
8.4.7 Working with objects 158
8.4.8 Working with enumeration types, compound types, contents, and dictionaries 160
8.4.9 Accessing the Fabasoft Folio Runtime 163
8.4.10 Accessing the transaction context 164
8.4.11 Tracing in Java 170
8.4.12 Support of old style Java implementation 170
8.4.13 Working with type definition of a customization point 170
8.5 Overriding an existing use case implementation 171
8.6 Use case wrappers 171
8.7 Use case wrappers (old style) 172
8.8 Implementing triggers 173
8.8.1 Object-level triggers 173
8.8.2 Property-level triggers 176
9 app.ducx Organizational Structure Language 180
9.1 Defining a position 181
9.2 Defining an organizational unit 181
9.3 Extending an organizational unit 182
9.4 Defining an access type 182
10 app.ducx Business Process Language 183
10.1 Defining a process 184
10.2 Defining an activity 184
10.2.1 Defining the actor 185
10.2.2 Defining the steps to be executed 186
10.2.3 Extending an activity 189
10.3 Defining conditions, case statements and loops 190
10.3.1 Defining a condition 190
10.3.2 Defining a case statement 191
10.3.3 Defining a loop 192
10.3.4 Defining a gateway 192
10.4 Defining parallel activities 193
10.4.1 Defining a block of parallel activities 194
10.4.2 Defining activity sequences within a parallel block 194
10.5 Defining sub processes 195
10.6 BPMN2 Modeling with app.ducx 198
11 app.ducx Customization Language 200

11.1 Customization points 200
11.1.1 Defining domain types 201
11.1.2 Generic method implementation for customization points 201
11.2 Customizations 202
11.2.1 Using domain types 202
11.2.2 Deprecated: Using add and override of a software solution or software edition 203
11.3 Using customizations 203
11.3.1 Concise example 204
11.4 Predefined customization points 204
11.4.1 PreGUI 204
11.4.2 InitWithState 205
11.4.3 PostGUI 206
11.4.4 IncreaseOrdinal 206
11.4.5 FormatValue 207
11.4.6 MetaParticipant 208
11.4.7 ACLConfiguration 209
11.4.8 ImageTypeConfiguration 210
11.4.9 ContentConfiguration 211
11.4.10 CPQuickSearchAction 211
11.4.11 CPQuickSearchAppearance 212
11.4.12 CPQuickSearchSuffix 212
11.4.13 CPSymbols 213
11.4.14 CPValidationExpression 214
11.4.15 CPContextExpressions 215
11.4.16 CPDocStateValidateConfig 215
11.4.17 CPRestrictClasses 216
11.4.18 NameBuild 217
11.4.19 FilterDispViewListAction 217
11.4.20 AggregationOverride 218
11.4.21 InsertActivityDef 219
11.4.22 GetLogoContainer 219
11.4.23 CreatePortalConfiguration 220
11.4.24 CPAllowedAttrDef 220
11.4.25 ListOption 221
11.4.26 ProcessOwnership 222
12 app.ducx Expression Language 223
12.1 General remarks concerning app.ducx expression language 223

12.1.1 Evaluating expressions at runtime 223
12.1.2 Testing expressions 224
12.1.3 Tracing in app.ducx expression language 224
12.2 Scopes 225
12.3 Types 226
12.4 Operators 227
12.4.1 Assignment operators 227
12.4.2 Logical operators 228
12.4.3 Calculation operators 228
12.4.4 Comparison operators 230
12.4.5 Conditional operator 232
12.4.6 Selection operator 232
12.4.7 $ operator 233
12.4.8 # operator 233
12.5 Predefined variables and functions 234
12.5.1 Predefined variables 234
12.5.2 Implicit properties of the STRING data type 234
12.5.3 Implicit properties of the DATETIME data type 235
12.5.4 Implicit pseudo functions 235
12.5.5 Working with contents and dictionaries 236
12.5.6 Getting the data type of an expression 236
12.5.7 String functions 237
12.5.8 List functions 239
12.5.9 Mathematical functions 239
12.5.10 Escape sequences for special characters 240
12.6 Getting and setting property values 241
12.7 Invoking use cases 242
12.8 Calculated identifiers 243
12.9 Accessing the transaction context 243
12.9.1 Working with transaction variables 244
12.10 Structured programming with expressions 244
12.10.1 Conditions 244
12.10.2 Loops 245
12.10.3 Raising an error 246
12.10.4 Error handling 247
12.10.5 Creating new transactions or opening a transaction scope 248
12.10.6 Returning values 248

12.10.7 Directives 249
12.11 Searching for objects – app.ducx Query Language 250
12.11.1 Options 251
12.11.2 Properties 252
12.11.3 Classes 253
12.11.4 Condition 253
12.11.5 Search query examples 254
12.12 Grammar of the app.ducx Expression Language 254
12.13 Grammar of the app.ducx Query Language 256
13 Testing and Debugging 257
13.1 Tracing in Fabasoft app.ducx projects 257
13.2 Display Trace messages in Eclipse 258
13.3 Coverage 259
13.3.1 Expression Coverage 260
13.3.2 Model Coverage 260
13.4 Debugging Fabasoft app.ducx projects 260
13.4.1 Debugging the Java implementation of an Fabasoft app.ducx project 260
13.4.2 Debugging the Fabasoft app.ducx Expression implementation of an app.ducx project 262
13.5 Expression Tester 264
13.6 Testing use cases with unit tests 265
13.7 Testing use cases with Fabasoft app.test 265
13.8 Fabasoft app.telemetry 266
14 Appendix 266
14.1 Comprehensive Java example 266
14.2 Creating a wizard 268
14.3 Not translated object classes 270
14.4 Transformation of legacy software components 271
14.4.1 Object Model Language 271
14.4.2 Resource Language 272
14.4.3 User Interface Language 272
14.4.4 Use Case Language 272
14.4.5 Organization Structure Language 273
14.4.6 Business Process language 273
14.4.7 Customization Language 273
14.4.8 Expression Language 273
14.5 Base App Profile Restrictions 273
14.5.1 Object Model Language 273

14.5.2 User Interface Language 274
14.5.3 Use Case Language 275
14.5.4 Organization Structure Language 275
14.5.5 Business Process language 275
14.5.6 Customization Language 276
14.6 Cloud App Profile Restrictions 276
14.6.1 Object Model Language 276
14.6.2 User Interface Language 277
14.6.3 Use Case Language 278
14.6.4 Organization Structure Language 278
14.6.5 Business Process language 278
14.6.6 Customization Language 279
14.7 Enterprise App Profile Restrictions 279
14.7.1 Object Model Language 279
14.7.2 User Interface Language 280
14.7.3 Use Case Language 281
14.7.4 Business Process language 281
14.7.5 Customization Language 281
15 Glossary 282
16 Bibliography 282

1 Introduction

Fabasoft app.ducx is Fabasoft's powerful use case-oriented application development system for developing composite content applications (CCAs). It is comprised of several declarative modeling languages that should be familiar to C++, C# and Java programmers. Fabasoft app.ducx combines the productivity gains of rapid application development languages and the power of Fabasoft Folio technology.

1.1 Fabasoft's software product portfolio

Fabasoft app.ducx is part of Fabasoft's software product portfolio. Combined with Fabasoft Folio, Fabasoft Mindbreeze Enterprise and Fabasoft app.telemetry, it provides everything you need to develop robust composite content applications quickly and efficiently.

1.1.1 Fabasoft app.ducx

Fabasoft app.ducx is the use case-oriented content application development platform for Fabasoft Folio. With Fabasoft app.ducx, the promise of rapid and easy CCA development becomes a reality.

Fabasoft app.ducx has been specifically designed to cover your application life cycle management (ALM) needs and supports you throughout the entire software development life cycle. The efficient implementation of CCAs is facilitated by domain-specific languages.

1.1.2 Fabasoft Folio

Fabasoft Folio is the out-of-the-box software for content governance and business process management.

Fabasoft Folio Enterprise

Fabasoft Folio Enterprise is the enterprise content management platform for enterprise applications tailored to your business needs.

The focus of Fabasoft Folio Enterprise is on customizing extensible and scalable CCAs providing powerful and comprehensive business process management.

Implementing your business processes with Fabasoft Folio Enterprise leads to improved quality, productivity, speed and security in your organization, and provides for deadlines, automatic exception handling, and process path optimization. A sophisticated access rights system safeguards your sensitive business data from unauthorized access.

- Business process management
- Process-oriented access control
- Easily customizable, extensible and scalable
- Support for up to 100,000 users and distributed, heterogeneous environments
- Connecter to iArchiveExchange, iArchiveLink and iArchiveSharePoint

Fabasoft Folio Enterprise is Fabasoft's comprehensive base edition covering enterprise content management and business process management requirements.

Fabasoft Folio Compliance

Fabasoft Folio Compliance is focused on providing compliance management and online archiving of objects, content and versions. Data is written transparently for users on cost-effective and auditable archive-media. Independently from the chosen archive media Fabasoft Folio Compliance enables

single instance storage of content by using a high performance resilient content store. This results in huge cost reductions for administration, backup and restore as well as operations management.

Archived documents and metadata can be restored into the production system according to the configured access rights. Besides the restoring of documents Fabasoft Folio Compliance supports the re-use of these data within your system for further processing. Due to the automatic version management complete traceability of changes is granted.

- Archive objects, versions and contents
- Single instance content store
- Transfer content and metadata to cost-effective archive media
- Content lifecycle management and retention periods
- Full-text search in archives with Fabasoft Mindbreeze Enterprise
- Restore archived contents into the production system

Fabasoft Folio Compliance covers the whole functionality range of Fabasoft Folio Enterprise extended with a sophisticated archiving system. It enables you to manage your content in a verifiable way.

Fabasoft Folio Governance

Fabasoft Folio Governance complements Fabasoft Folio Compliance by electronic records management (ERM). It is Fabasoft's MoReq2 certified records management solution for organizations having to establish powerful corporate governance based on consistent and orderly business records.

MoReq2 is a European specification for records management (http://www.moreq2.eu) respecting international norms and standards. In conjunction with a comprehensive framework for testing software products it can be proofed if a records management solution meets the specified requirements.

- Electronic records management
- Corporate governance
- MoReq2 certified

1.1.3 Fabasoft Mindbreeze Enterprise

With the seamless integration of Fabasoft Mindbreeze Enterprise into Fabasoft Folio, your CCA is automatically search-enabled, and benefits from an impressively powerful enterprise search software solution. The Fabasoft Mindbreeze Enterprise platform focuses on the meaning and relevancy of information and treats the answers to questions as actionable information, and not just a collection of links. Fabasoft Mindbreeze Enterprise supports relevancy models that are applied to specific types of queries and helps users to quickly find the exact, most relevant information.

1.1.4 Fabasoft app.telemetry

Fabasoft app.telemetry provides powerful and easy to use tools for state-of-the-art applications management. As demands on IT services operations dramatically increase to provide high availability and low response time service level agreements, more than a classic system management infrastructure is required. With its tight and efficient integration with Fabasoft software products, Fabasoft app.telemetry is the unrivaled answer for applications management.

1.2 Benefits of creating CCAs based on Fabasoft technology

In this section, the main benefits of developing CCAs based on Fabasoft software products are presented and discussed briefly.

1.2.1 Content governance

CCAs built with Fabasoft app.ducx benefit from Fabasoft Folio's seamless integration of business process management, enterprise content management, and records management. The advantage for your business arises from the widespread and consistent availability of information and the standardized, secure way it can be accessed in the context of your CCA.

Using Fabasoft Folio as its basis, your CCA also meets all of your knowledge management, relationship management and compliance management needs.

1.2.2 Single-instance content store

Fabasoft Folio maintains content in a high-performance single-instance content store. This means that data used by your CCA is always available as required – while saving expensive disk space and simplifying content management by just keeping one copy of your content within your entire organization.

1.2.3 Compliance

CCAs based on Fabasoft Folio enable you to rapidly and easily transform your compliance procedures and guidelines into structured electronic business processes. The system maintains revision-safe documentation of the implementation of these processes and all related work. This documentation is available to authorized individuals for consultation during compliance audits.

1.2.4 Content life cycle management

Fabasoft Folio offers complete management of your CCA's business objects for the entire duration of their life cycle.

Approved text modules and templates are used right from the start when the document is first created. Documents are clearly ordered according to business connections, events, projects, files or any other relationship meaningful to your business. Access rights ensure that security guidelines for sensitive documents such as personnel records are complied with while at the same time guaranteeing maximum flexibility of business processes.

Fabasoft Folio features a powerful workflow technology to achieve efficiency and transparency in business processes. Ad hoc and predefined workflows can be easily created, configured and re-used. Using a graphical user interface, the status of any given process and its associated business objects can be monitored in real time.

1.2.5 Access protection and access rights system

A sophisticated access rights system ensures that sensitive data can only be viewed by authorized users. Who is authorized to do what in which phase of a process can be specified effortlessly, right down to individual operations and data fields. In addition, an automatic record can also be kept of accessed content that requires special protection.

1.2.6 Fabasoft reference architecture

The Fabasoft reference architecture offers your CCAs a resilient backbone that can scale to support from 100 to 100,000 users. In conjunction with Fabasoft app.telemetry this allows you to achieve and manage your defined application service levels.

1.2.7 Platform independence

Fabasoft's line of software products is available on both Linux and Microsoft Windows platforms to meet the evolving requirements of your business.

Fabasoft Folio offers you several choices: Linux in the datacenter and Microsoft Windows on the desktop? A full open source stack in the datacenter and on the desktop? A pure Microsoft Windows environment?

Fabasoft Folio is an open system and can be used as a framework for CCAs based on Java. These CCAs are developed using Fabasoft app.ducx in your Eclipse development environment.

1.3 Benefits of Fabasoft app.ducx

Fabasoft app.ducx relies on cutting-edge technology to facilitate rapid application development. This section gives a concise overview of the main benefits of Fabasoft app.ducx. For more detailed information, please consult [Faba10a].

1.3.1 Shorter time to value

The use case-oriented approach of developing CCAs with Fabasoft app.ducx focuses on the perspective of the end-user. Powerful domain-specific languages reduce the complexity of projects leading to less ambiguity and fewer errors between specification and implementation. As a result, maintenance and development costs can be reduced while at the same time increasing end-user satisfaction.

Automated testing with Fabasoft app.test integrated into your development environment will help you to achieve a consistent level of high quality application development. Through its intuitive and easy-to-use interface, running comprehensive tests against your CCA has never been easier.

1.3.2 Holistic application life cycle management

Software development with Fabasoft app.ducx follows a development process model that describes a standardized approach to the variety of tasks and activities that take place during the process of creating a CCA. This development process model is called the software development life cycle.

Fabasoft app.ducx is part of the Fabasoft application life cycle management suite that supports the entire software development life cycle. As illustrated by the next figure, Fabasoft app.ducx allows you to manage and control the software development life cycle from within your development environment (Eclipse).

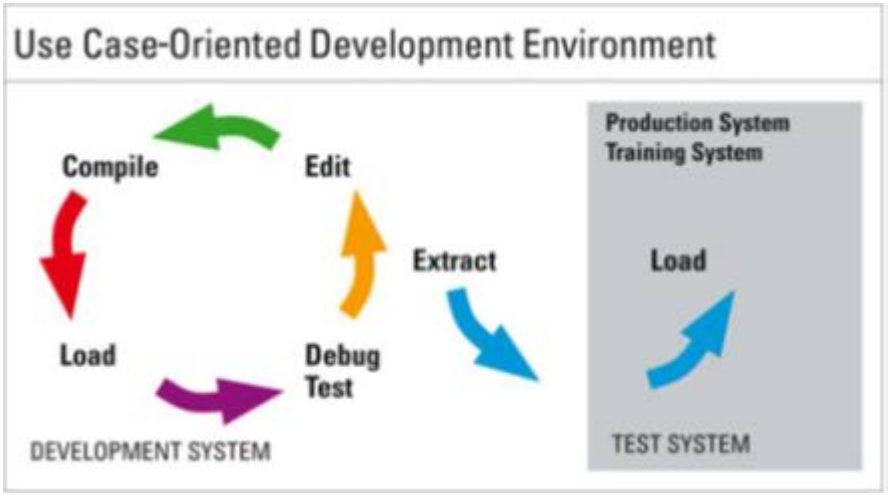


Figure 1: Software development life cycle with Fabasoft app.ducx

1.3.3 Integrated in your development environment

In keeping with Fabasoft's commitment to provide platform independence, Fabasoft app.ducx is a development platform for both the Microsoft Windows and Linux worlds and is therefore integrated into Eclipse. This means greater flexibility for your organization while developing solutions.

1.3.4 Automated testing

Fabasoft app.test is a powerful tool for the automated testing of your CCA. Recording and playing of interactions happens within a context-sensitive user interface.

1.3.5 Localization and customization

Fabasoft app.ducx allows for an easy and effortless localization and customization of your CCA.

1.3.6 Integrated compiler

Fabasoft app.ducx allows you to benefit from the rich metadata, compile-time syntax checking, static typing and auto-completion previously available only to imperative programming languages.

Using the compiler provided with Fabasoft app.ducx, a Fabasoft app.ducx project can be compiled by simply triggering the build process of the respective development environment, which produces a Fabasoft Folio container file (with a `.coo` extension) and – if the project also contains use case implementations in Java – a Java archive (`.jar`) file. These files are then automatically packaged, compressed, and loaded into the Fabasoft app.ducx domain.

1.3.7 Delta loader

The Fabasoft app.ducx compiler produces a Fabasoft Folio container file, which is uploaded to a web service connected to your Fabasoft app.ducx domain. In some projects, this Fabasoft Folio container file holding your software component may get quite large. To optimize upload performance, Fabasoft app.ducx uses a built-in delta loader which, whenever possible, only uploads the changes since the last upload in order to save time and network bandwidth.

1.3.8 Automatic address management

With Fabasoft app.ducx, you do not need to worry about address management anymore. Fabasoft app.ducx automatically assigns addresses to all component objects created and keeps track of address assignments.

1.3.9 Fabasoft app.ducx cache

Fabasoft app.ducx maintains a local cache on your development computer, which contains all the information necessary for compiling an app.ducx project without needing to connect to the development web service.

When you reuse functionality provided by another software component in your Fabasoft app.ducx project, you have to add a reference to this software component, which triggers an update of the local Fabasoft app.ducx cache.

1.3.10 "Convention over configuration" principle

Fabasoft app.ducx uses a set of coding and naming conventions that when adhered to eliminate the need for configuring every single aspect of your software solution. As a result, you need to write less code which will make software development and maintenance much easier.

1.4 Who should read this book?

This book is aimed primarily at software developers interested in exploring the tremendous potential of Fabasoft app.ducx.

However, you do not need to be a professional programmer to create your own cross-platform applications for Fabasoft Folio since Fabasoft app.ducx makes developing software components easy.

As shown in the following chapters, all you need to get started is a basic understanding of a few technologies that are familiar to most developers who have already implemented software components for Fabasoft Folio in the past.

This book assumes that the reader has some level of familiarity with Fabasoft Folio, Eclipse and Java. Reading this book in conjunction with other books that are devoted specifically to these topics may be useful if you are not already comfortable using these technologies.

For your convenience, concepts and technologies that are new in Fabasoft app.ducx are explained in great detail throughout the book.

1.5 General remarks concerning examples

The examples used throughout this book contain code fragments that were specifically created as examples to highlight the use of a particular concept or aspect of software development with Fabasoft app.ducx.

Please be aware that not all examples in this book are completely self-contained. In order to save space, certain parts have been omitted in some of the examples.

2 Installing Fabasoft app.ducx

This chapter explains how to install and configure Fabasoft app.ducx for use on your system.

2.1 Software requirements

Fabasoft app.ducx must be installed in the Fabasoft Folio Domain that is used for development. The Fabasoft Folio Domain for development must have the same version as the productive Fabasoft Folio Domain where the developed software components are installed.

The Fabasoft Folio Domain can also be installed on a remote computer. Only the Fabasoft app.ducx plug-in has to be installed on the developer's computer as Fabasoft app.ducx does not require the Fabasoft Folio Kernel. All information required by Fabasoft app.ducx is retrieved through its service-oriented architecture. Thus, it is still possible to continue development if the connection to the development web server becomes unavailable.

It is recommended that the Fabasoft app.ducx plug-in has the same version as the Fabasoft Folio Domain, but it is supported to use a newer Fabasoft app.ducx plug-in with an older Fabasoft Folio Domain.

For more information on Fabasoft app.ducx, please consult [Faba10a].

2.2 Licenses

At least two licenses are necessary for a Fabasoft Folio Developer Domain. One license is required for the Fabasoft Folio Domain itself. The other license or licenses contain Fabasoft app.ducx that is used to specify the mayor and minor domain ID of the component objects created during the development process. Make sure that the Fabasoft app.ducx license is loaded into the Fabasoft Folio Domain before starting developing.

2.3 Which software do I need for using Fabasoft app.ducx?

A popular development environment is currently supported by Fabasoft app.ducx:

- Eclipse 4.2.2 with
- Oracle Java SE Development Kit 7 Update 13 (JDK)
- Apache Ant 1.8.3

Fabasoft app.ducx does not require a local installation of a Fabasoft Folio Domain. Only the lightweight Fabasoft app.ducx plug-in has to be installed on the developer's computer.

Note: For detailed information on supported operating systems and software see the Readme file in the root directory on the Fabasoft product CD or product DVD.

2.4 Using Eclipse

Eclipse can be downloaded free of charge from the Eclipse web site [Ecli10] and the Java Development Kit can be obtained from the java.sun.com web site [Orac10a].

Eclipse uses so-called features to package plug-ins and to allow for full integration with the Eclipse platform. Fabasoft app.ducx provides a feature that must be installed before it can be used with Eclipse.

2.4.1 Installing the Fabasoft app.ducx feature using the Update Site

You can install the Fabasoft app.ducx feature by opening the "Help" menu and clicking "Install New Software". Click "Add". In the following dialog input an individual name (e.g. "app.ducx Update Site") and the location "http://update.appducx.com".

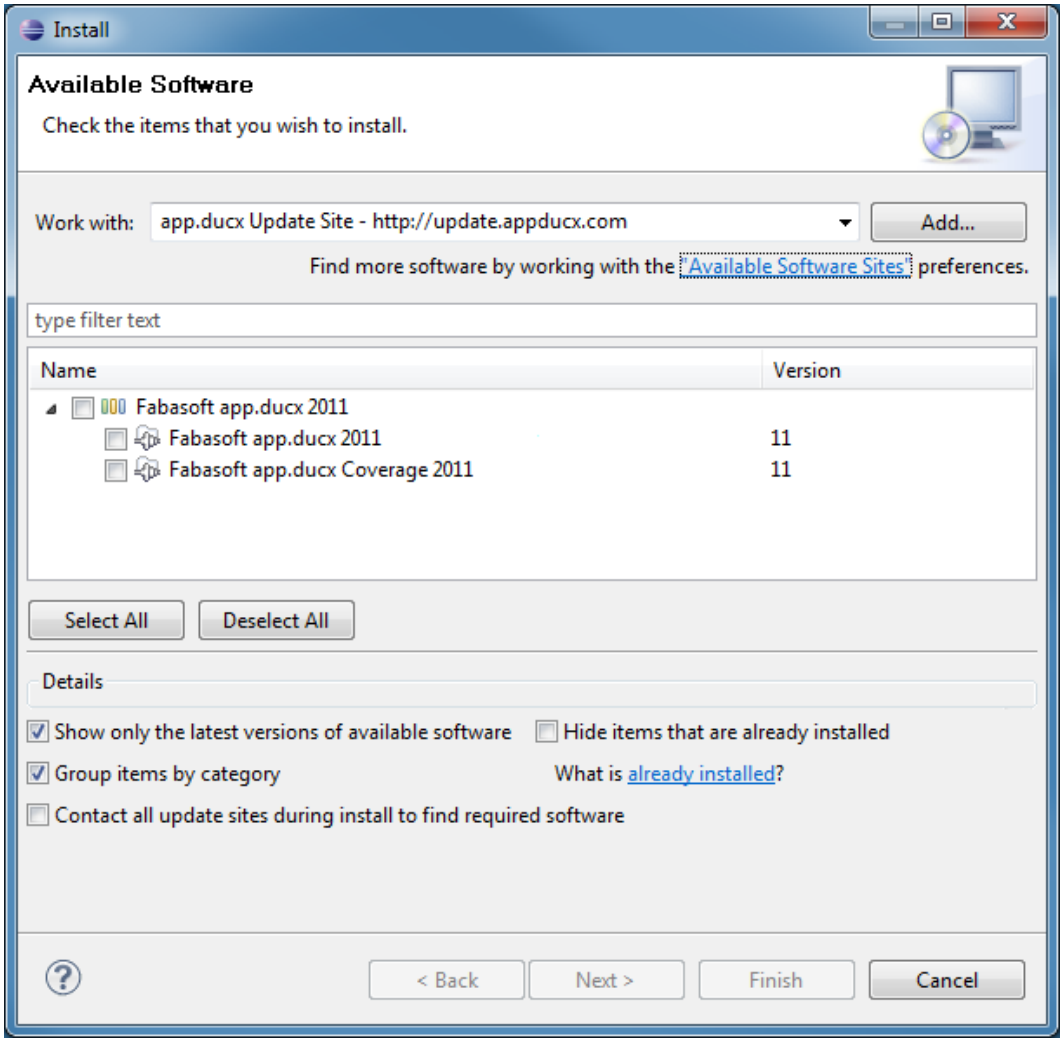


Figure 2: Specifying a new update site in Eclipse

Select the “Fabasoft app.ducx” feature and click “Next”. Click “Finish”.

Note: If your Eclipse installation does not meet all prerequisites, missing features are downloaded from the Eclipse Update Site. Make sure that thise site is activated in the “Available Software Sites”.

2.4.2 Installing the Fabasoft app.ducx feature using the archive

You can also install the Fabasoft app.ducx feature using an archive. Open the add dialog in the install wizard. In the following dialog click “Archive”. Select the Fabasoft app.ducx archive provided on the Fabasoft app.ducx DVD (`Setup\ComponentsAppducx\Package\ETC\ducx.eclipse.updatesite.zip`) and click “OK”.

Select the “Fabasoft app.ducx” feature and click “Next”. Click “Finish”.

Note: If your Eclipse installation does not meet all prerequisites, missing features are downloaded from the Internet. Make sure that your proxy server (if necessary) is configured correctly in Eclipse (“Window” > “Preferences” > “General” > “Network Connections”) and that you are connected to the Internet.

2.4.3 Updating the Fabasoft app.ducx feature

Every time the Fabasoft Folio Domain that is used for development is updated also the Fabasoft app.ducx feature for Eclipse has to be updated. The update process is the same as described in chapter 2.4.1 “Installing the Fabasoft app.ducx feature using the Update Site”.

Note: How to update Fabasoft app.ducx projects for a new software release is described in chapter 3.4 “Updating Fabasoft app.ducx projects using Eclipse”.

2.4.4 Improving the performance of the Fabasoft app.ducx compiler

To improve the performance of the Fabasoft app.ducx compiler, edit the `eclipse.ini` file found in the Eclipse installation folder and assign sufficient memory to the Java virtual machine by setting the `-Xms` and `-Xmx` parameters to an adequate value (e.g. “768m” or “1g” for the 32-bit edition of Eclipse, “1500m” or higher for the 64-bit edition). When using the 64-bit edition of Eclipse, it is also advisable to include the `-XX:+UseCompressedOops` option. For improved garbage collector performance also add the `-XX:+UnlockExperimentalVMOptions` and `-XX:+UseG1GC` options.

Example

```
-showsplash
org.eclipse.platform
--launcher.XXMaxPermSize
256m
-vmargs
-Xms256m
-Xmx768m
-XX:+UseCompressedOops
-XX:+UnlockExperimentalVMOptions
-XX:+UseG1GC
```

2.4.5 Using Fabasoft app.ducx with an older Fabasoft Folio Domain

2.4.5.1 Fabasoft Folio Domain

Do not install the current version of Fabasoft app.ducx. Let the Fabasoft Folio Domain as it is.

2.4.5.2 Client

Install or update the current Fabasoft app.ducx plug-in for Eclipse. The app.ducx projects are compiled for the version and software edition of the Fabasoft Folio Domain.

Note:

- If the Fabasoft Folio Domain has the version 2008 Production or 8.0 Production, the target version for which the projects should be compiled has to be set manually in the `.ducxproject` file: `<ns:VersionTarget>VERSION8</ns:VersionTarget>`.
- Highlighting of keywords in the development environment is based on the current release.

3 General Use of Fabasoft app.ducx

This chapter focuses on the general use of Fabasoft app.ducx like creating Fabasoft app.ducx projects or working together on a single project.

3.1 General structure of a Fabasoft app.ducx project

A Fabasoft app.ducx project consists of one or more source files, formally known as compilation units, and project files that are specific to the development environment used (Eclipse). The source code for an app.ducx project is typically stored in one or more text files with a file extension of `.ducx-om`, `.ducx-rs`, `.ducx-uc`, `.ducx-ui`, `.ducx-os`, `.ducx-bp` or `.ducx-cu`.

Each source file must contain exactly one type of model. A model consists of a block that can contain import declarations and element definitions.

The following example illustrates the skeleton of a Fabasoft app.ducx object model source file that contains import declarations and some element definitions.

Example

```
//========================================================================
//
// Copyright information
//
//========================================================================
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class Product : BasicObject {
    mlname;
    string[] productdescription;
    currency unitprice;
  }
}
```

Note: The encoding of source files in an Fabasoft app.ducx project is UTF-8.

3.2 Creating Fabasoft app.ducx projects using Eclipse

With Eclipse, you can also either create a new Fabasoft app.ducx project from scratch or create a project from an existing software component.

Before creating your first Fabasoft app.ducx project, it is recommended that you define a default web service in the Eclipse preferences.

3.2.1 Fabasoft app.ducx-specific settings in the Eclipse preferences

Fabasoft app.ducx allows you to define several Fabasoft app.ducx-specific settings in the Eclipse preferences dialog box. To configure these settings, click *Preferences* on the *Window* menu, and select “Fabasoft app.ducx” in the dialog box (see next figure).

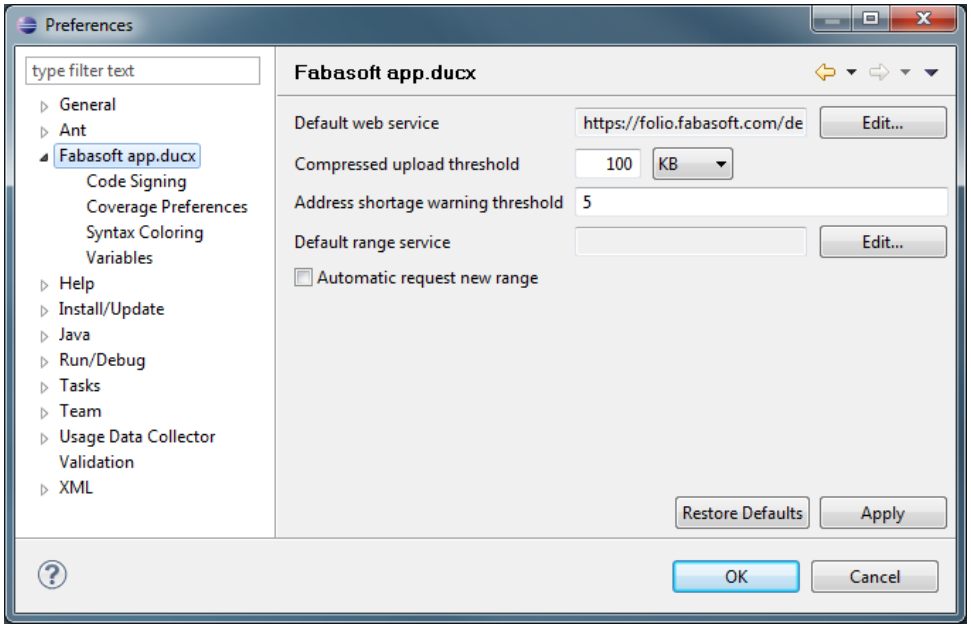


Figure 3: app.ducx-specific preferences in Eclipse

It is recommended that you define a *Default Web Service* for Fabasoft app.ducx in the Eclipse preferences before you create your first Fabasoft app.ducx project.

In the preferences dialog box, you can also set the threshold for compressed uploads. When running a Fabasoft app.ducx project, a software component is automatically generated from the output of the Fabasoft app.ducx compiler, and uploaded to the development web service. If the file size of the software component to be uploaded exceeds the threshold defined in the *Compressed Upload Threshold* field, the software component is automatically compressed to minimize the upload time.

You can also activate code signing and customize the colors used for syntax coloring in the preferences dialog box.

3.2.2 Creating a new Fabasoft app.ducx project

On the *File* menu, point to *New*, and then click *Project* to invoke the “New Project” dialog box (see next figure). In this dialog box, select “Fabasoft app.ducx project” from the “Fabasoft app.ducx” folder and click *Next*.

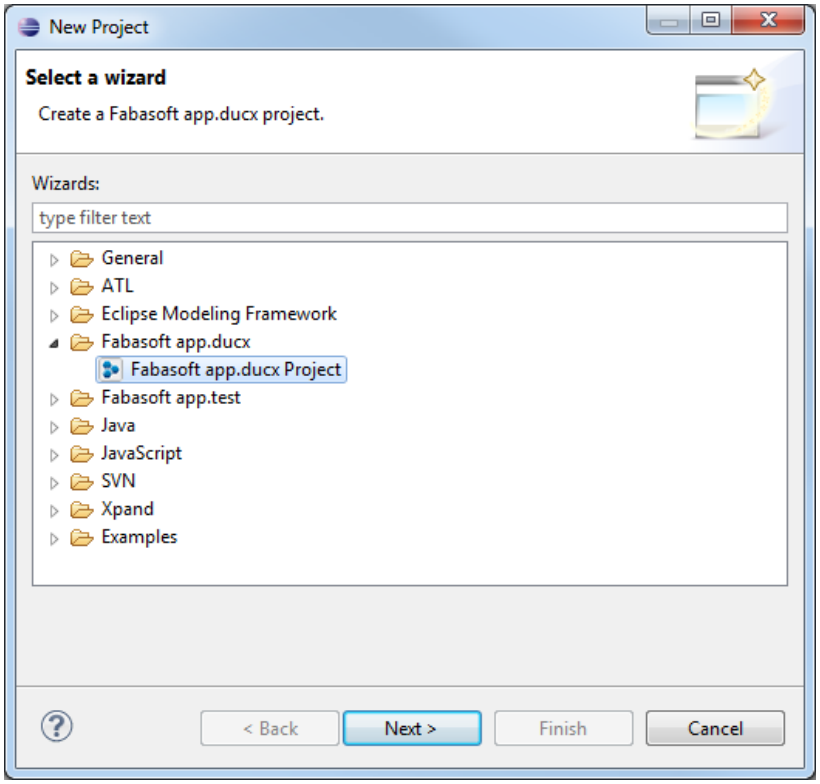


Figure 4: Eclipse New Project wizard

In the “Create a Fabasoft app.ducx project” dialog box (see next figure), enter a *Project Name* for your project.

If you clear the *Use default location* check box, you can enter a path for your new Fabasoft app.ducx project. Otherwise, the default location defined in the Eclipse preferences is used.

By default, the web service configured in the Eclipse preferences is used for connecting to your Fabasoft app.ducx domain. However, you may also define a project-specific web service by clearing the *Use default web service* check box.

In the *Software component* field, you have to provide the reference of the software component created for your new Fabasoft app.ducx project. Additionally select the domain ID the software component should belong to.

During the development process component objects are created. Each component object gets a unique address that contains a major and minor domain ID and a subsequent number. The domain ID is defined in the license file for Fabasoft app.ducx and cannot be changed.

Note: To decide which domain ID should be used, open the Fabasoft app.ducx license file with an editor and search for `COOSWCLM@1.1:domainmajorid` and `COOSWCLM@1.1:domainminorid`. The file name of the license file usually contains the domain ID, too.

For productive development additionally a personal address range file and an address range has to be entered in the *Address file* and *Address ranges* field to ensure unique addresses. Select a start address and an end address. The natural numbers within the range are used for the created component objects. If all addresses are used, an error is generated. In this case an additional range for the Fabasoft app.ducx project has to be provided.

Attention: All addresses have to be world-wide unique. On the one hand this is assured by Fabasoft, providing each major and minor domain ID combination only once. On the other hand the address ranges have to be managed by the customers themselves. If several developers work within a domain ID, each developer is only allowed to use defined address ranges such that no address can be assigned twice.

The current domain cannot be ranged and must not be used for productive development.

Note: Later on the address range file and the defined address ranges can be modified in the Fabasoft app.ducx project settings. An address range file can be added or changed and additional address ranges can be defined.

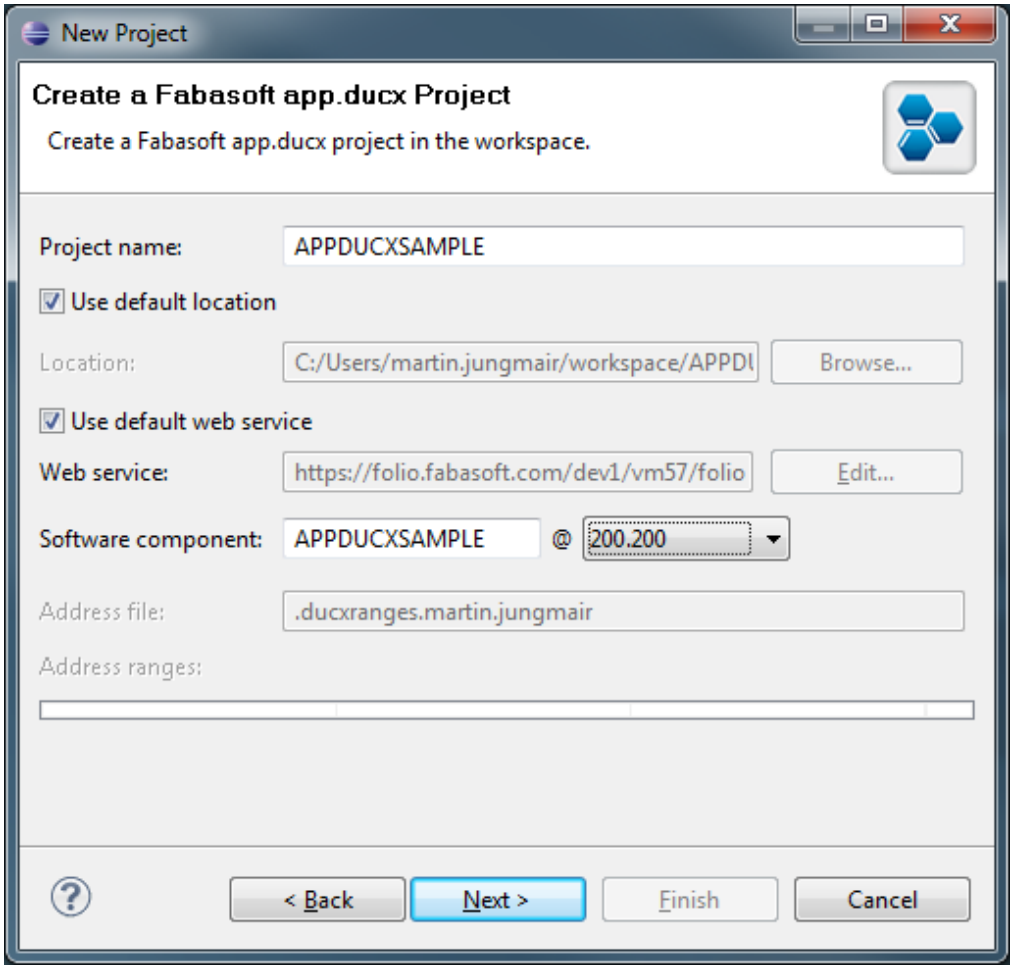


Figure 5: Fabasoft app.ducx project wizard in Eclipse

After clicking *Next*, you are taken to the dialog box pictured in the next figure, where you can add software component references to your Fabasoft app.ducx project. For detailed information on software component references, please refer to chapter 3.3.2 “Adding a software component reference”.

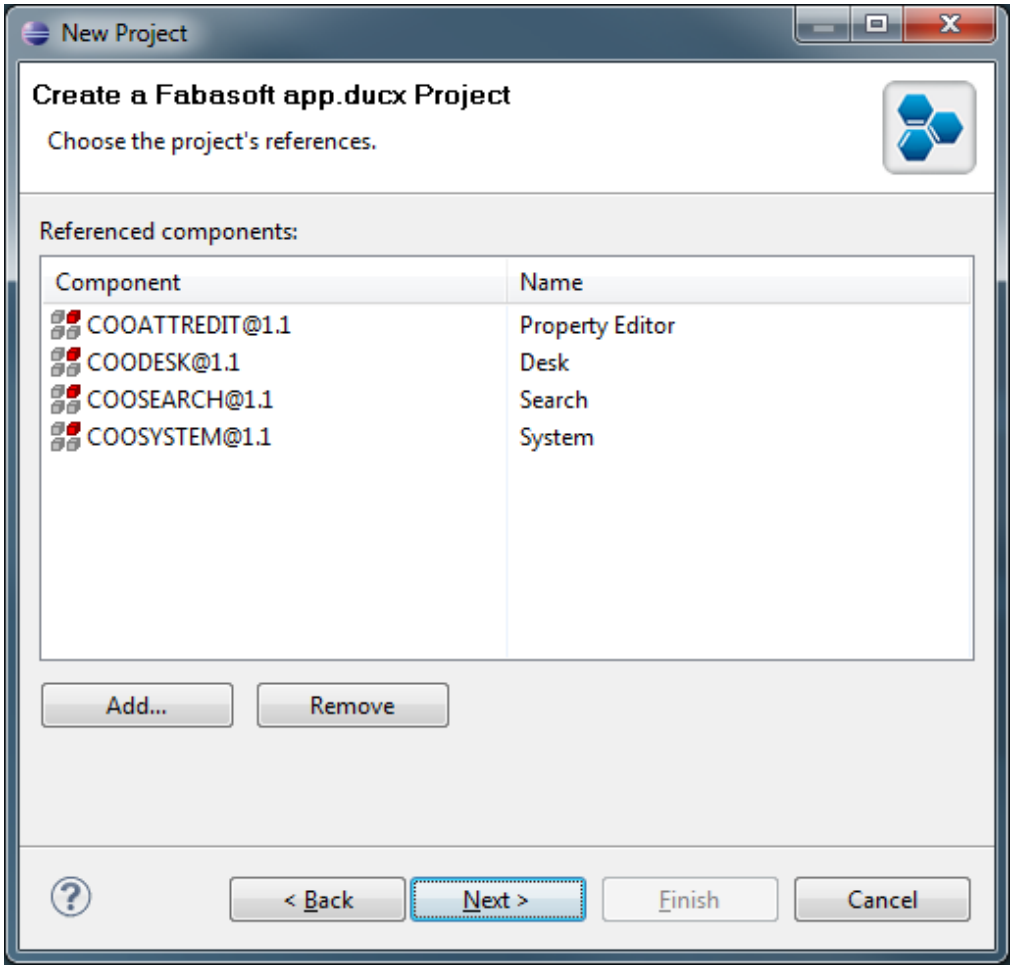


Figure 6: Defining the software component references for your project

Click *Next* for the wizard to take you to the next step, where you need to provide a *Name* and a *Version* for your software component along with a *Copyright file*. Confirm your choices by clicking *Next*.

This will take you to the dialog box shown in the next figure. In this dialog box, select *Enable Java support* if you want to use Java to implement use cases in your Fabasoft app.ducx project. If you select *Generate only for shortcuts*, Java classes are only generated for software components that are defined as shortcuts. Shortcuts for `COOSYSTEM@1.1` and the own software component are mandatory and created automatically. This way the compiling performance can be improved. *Strip unused symbols from output* should be used for a release build to reduce the size of the JAR file. No stripping of unused symbols is an advantage in the development process because the project compiles faster.

For detailed information on how to implement use cases in Java, please refer to chapter 8.4 “Implementing a use case in Java”.

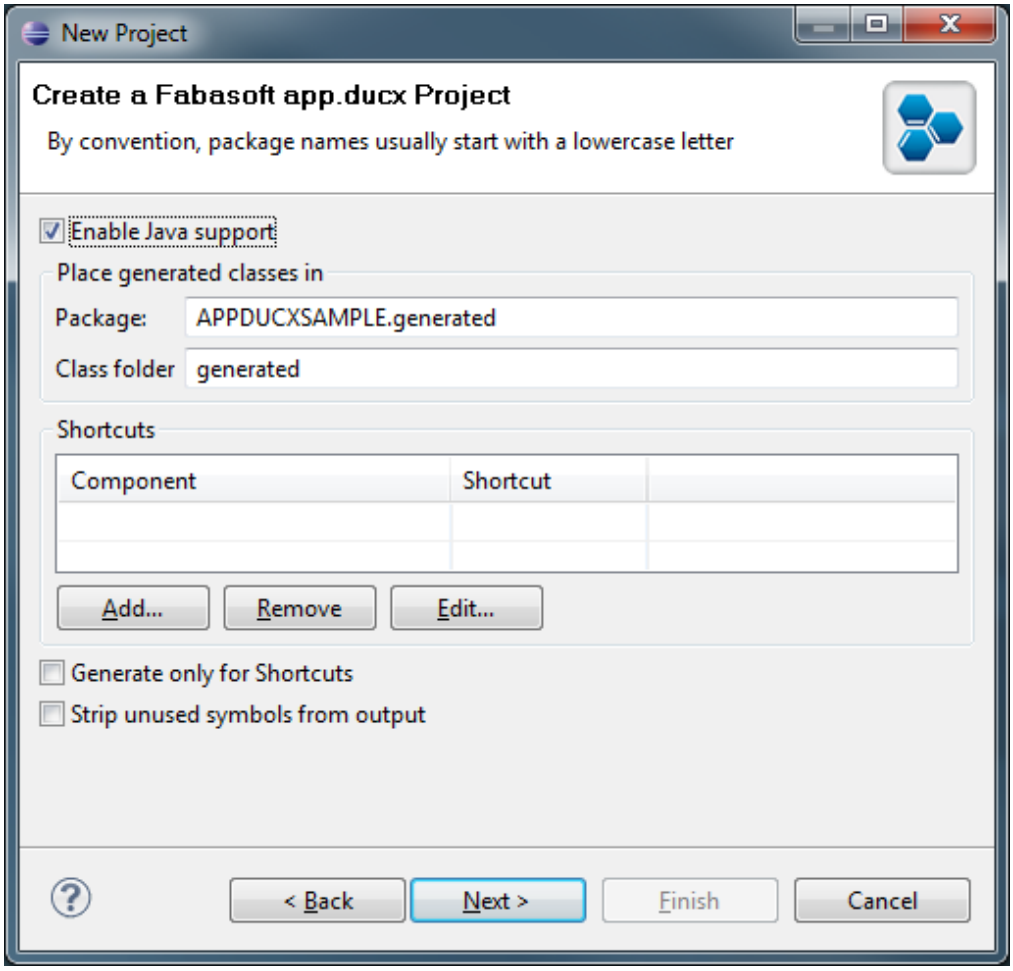


Figure 7: Enabling Java support for a Fabasoft app.ducx project

Click *Next* to get to the dialog box depicted in the next figure. In the dialog box, you may select template files that you want the wizard to automatically generate for your Fabasoft app.ducx project. For instance, as your Fabasoft app.ducx project will most likely contain object model elements, you may want to consider checking the `.ducx-om` template for the wizard to add an object model file to your app.ducx project for you. However, you can also add files to your Fabasoft app.ducx project later.

Click *Finish* for the wizard to create your new Fabasoft app.ducx project.

The Fabasoft app.ducx feature for Eclipse also includes a predefined perspective that is optimized for working with Fabasoft app.ducx. After clicking *Finish*, you are asked whether you want to activate the Fabasoft app.ducx perspective.

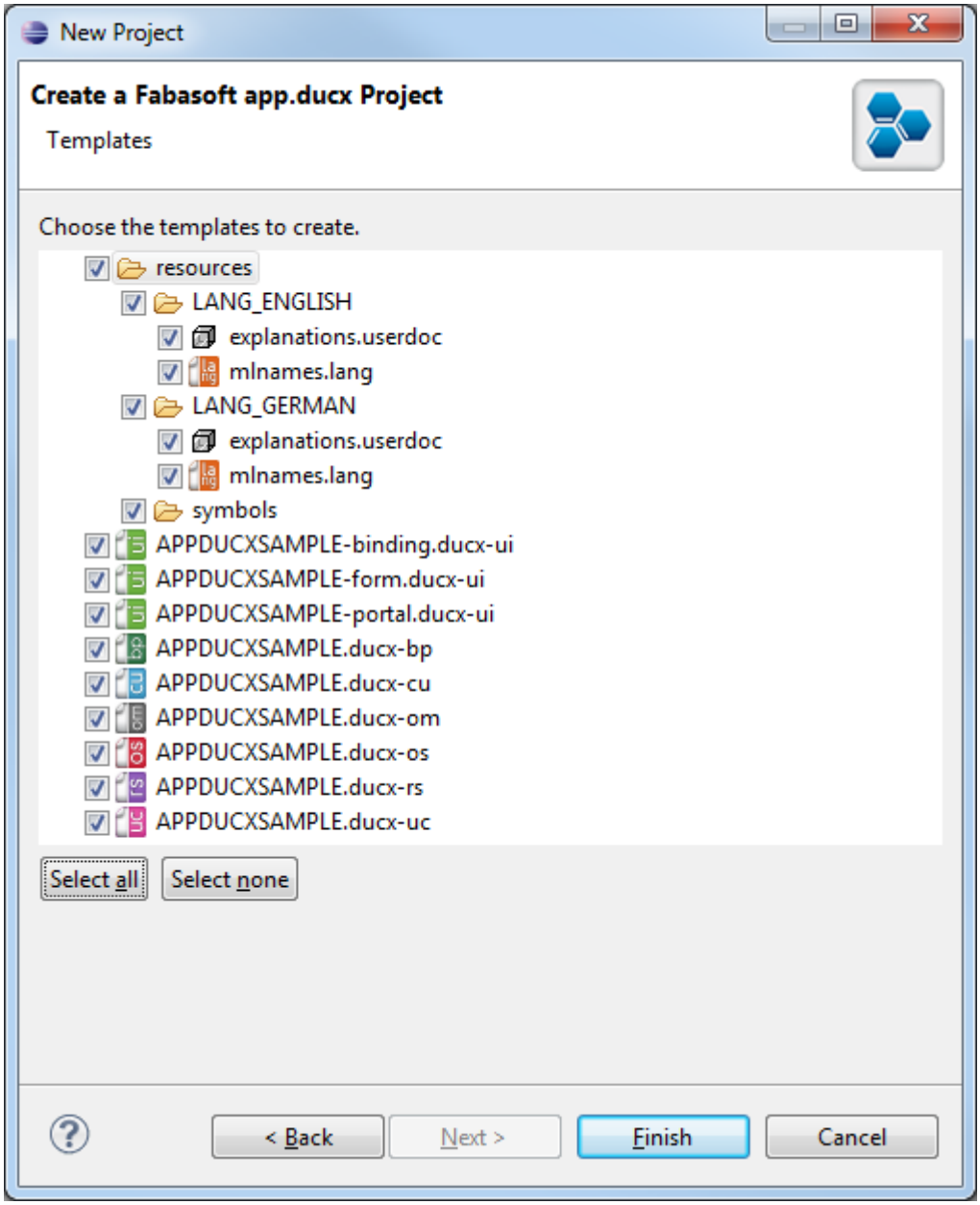


Figure 8: Selecting template files

3.2.3 Creating an Fabasoft app.ducx project from an existing software component

Fabasoft app.ducx allows you to create a Fabasoft app.ducx project from an existing software component. To do so, click *Import* on the context menu in Eclipse Project Explorer. Select “Existing Component as Fabasoft app.ducx Project” and click *Next*.

This will take you to the dialog box depicted in the next figure where you can select the source for creating the new Fabasoft app.ducx project. You may either select a container object (`.coo`) file from the file system or a software component from your development domain. Enter a *Project name* and the reference of your new software component in the *Component* field, and click *Next* to proceed.

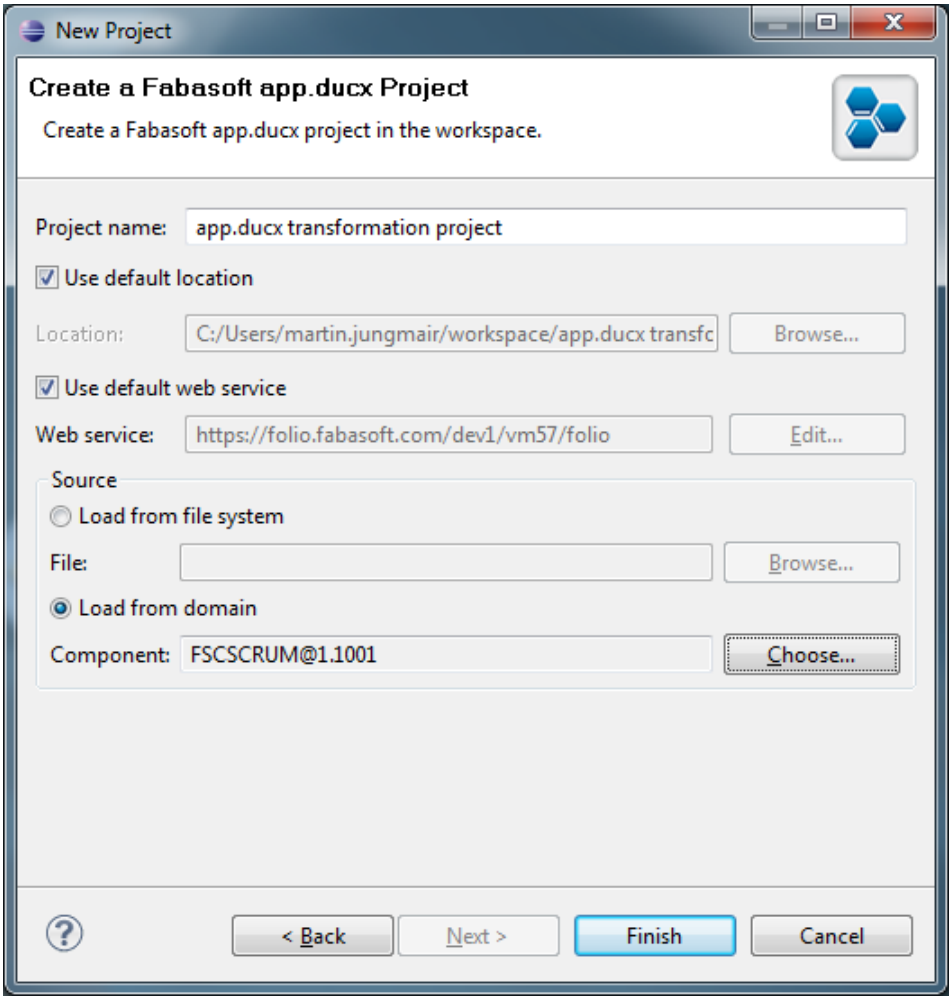


Figure 9: Creating a Fabasoft app.ducx project from an existing software component

The remaining steps of the wizard are similar to the steps when creating a new Fabasoft app.ducx project from scratch.

After completing the remaining steps, the wizard creates a Fabasoft app.ducx project based on the software component you selected as source, and transforms the component objects belonging to the software component into domain specific language source code.

Note: A full transformation of all elements of the software component used as source might not be possible. For a detailed overview of how each language construct transformed is please refer to chapter 14.3 “Not translated object classes”.

3.3 Working with Fabasoft app.ducx projects using Eclipse

3.3.1 Running a Fabasoft app.ducx project

You have to create a new launch configuration before you can run a Fabasoft app.ducx project in Eclipse. To do so, click *Run Configurations* on the *Run* menu. This will bring up the dialog box shown in the next figure.

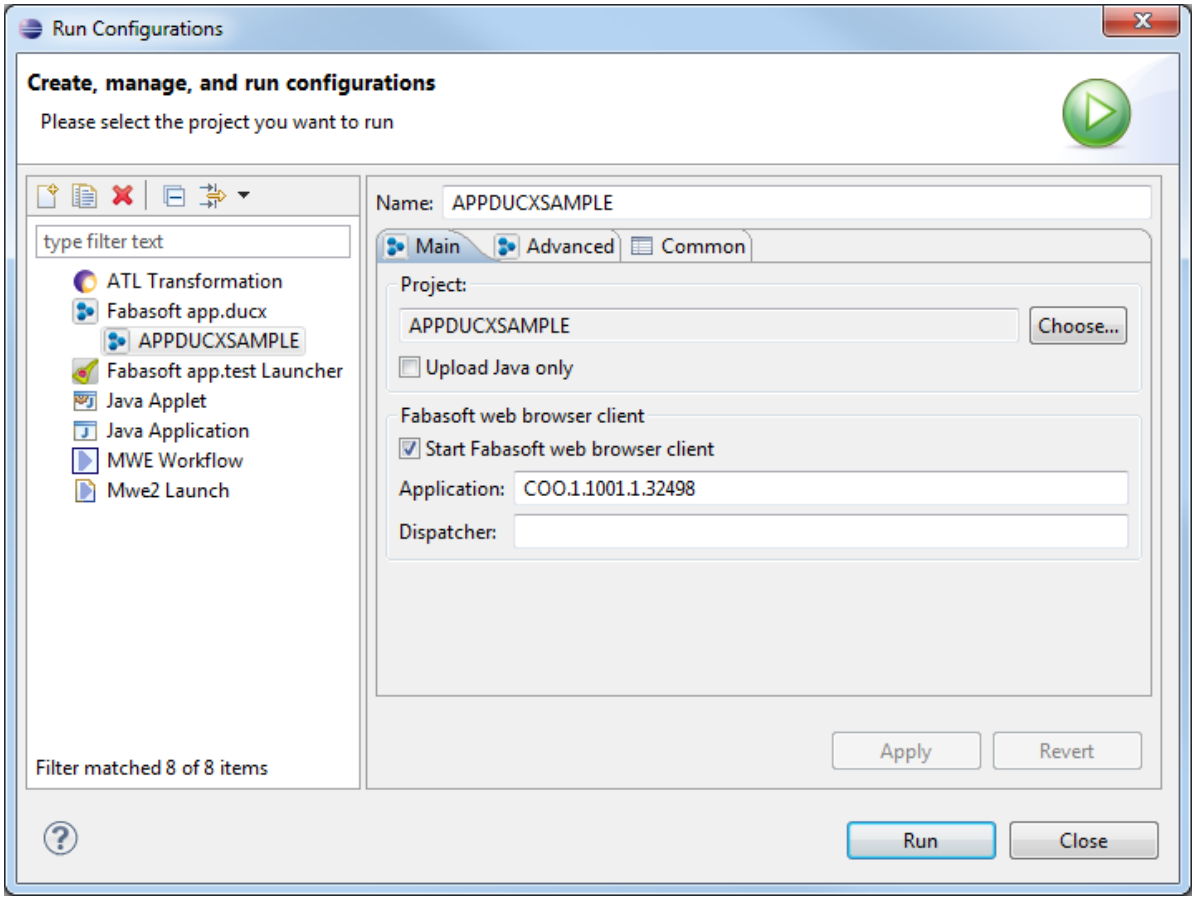


Figure 10: Creating a new launch configuration in Eclipse

In this dialog box, click the *New launch configuration* symbol and enter a *Name* for the new launch configuration. In addition to this, select the *Project* by clicking *Choose*. Click *Apply* to save your settings, and *Run* to run the Fabasoft app.ducx project.

Once a Fabasoft app.ducx launch configuration has been created, you can select the existing launch configuration on the *Run as* menu to run the Fabasoft app.ducx project.

Modified project files may be saved automatically depending on the launch configuration of Eclipse.

Note: A Fabasoft app.ducx project is always compiled for the current version and software edition of the Fabasoft Folio Developer Domain.

3.3.2 Adding a software component reference

Whenever you either explicitly or implicitly reuse parts of the functionality provided by another software component, you have to add a reference to this software component.

To add a software component to your Fabasoft app.ducx project in Eclipse, select *Add Reference* from the *Software Component References* context menu in the Eclipse Project Explorer. In the dialog box shown in the next figure, select the software component to add. You may also select more than one software component.

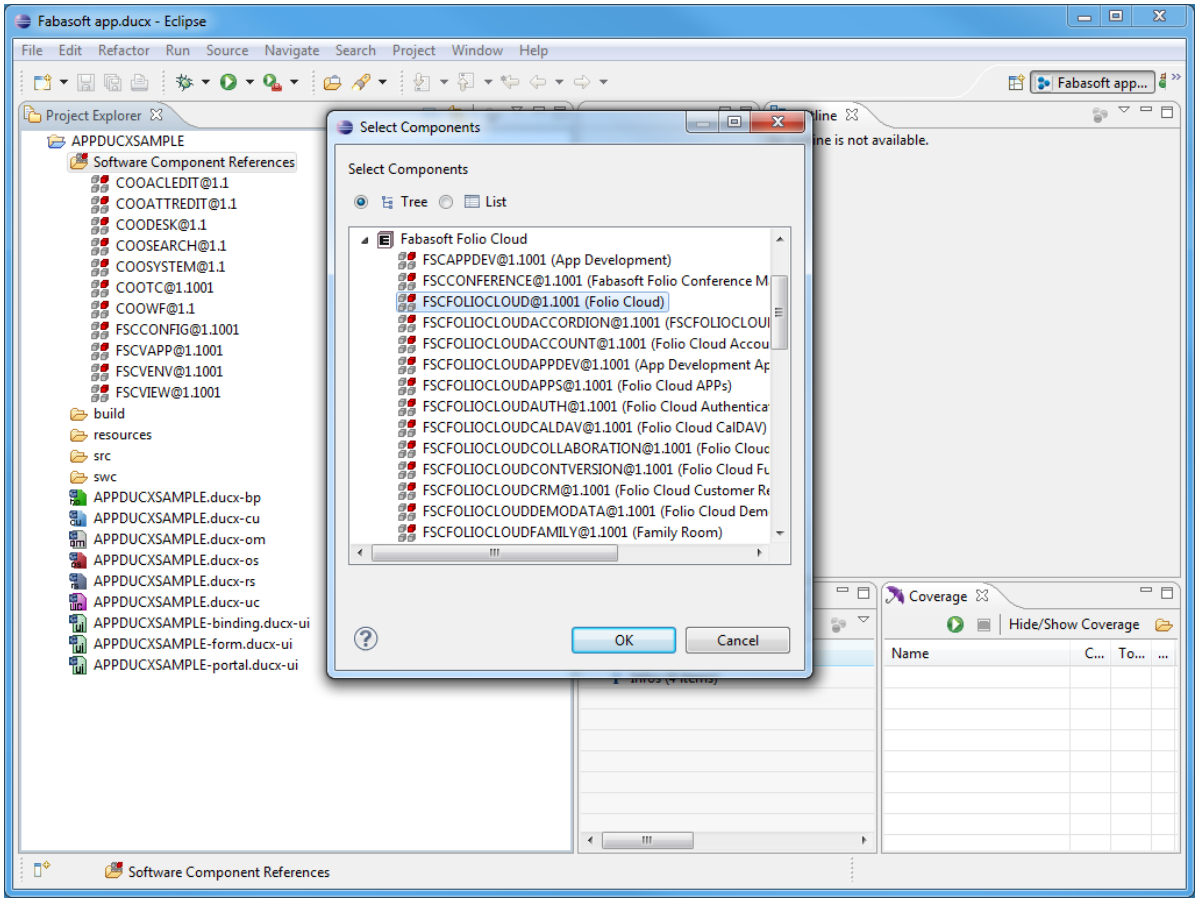


Figure 11: Adding a reference to a software component

Adding a reference to a software component triggers a web service call, which downloads required information about this software component to the local Fabasoft app.ducx cache on your computer. The cached information is then used by the Fabasoft app.ducx compiler and for providing context-sensitive help.

All added software components will be inserted in the list of prerequisite components of the software component when the project is built.

You can also remove references to unused software components at any time.

3.3.3 Adding a source file

In Eclipse, you can add new source files to an existing Fabasoft app.ducx project at any time by selecting the type of source file you would like to add from the *New* context menu of your Fabasoft app.ducx project in the Eclipse Project Explorer (see next figure).

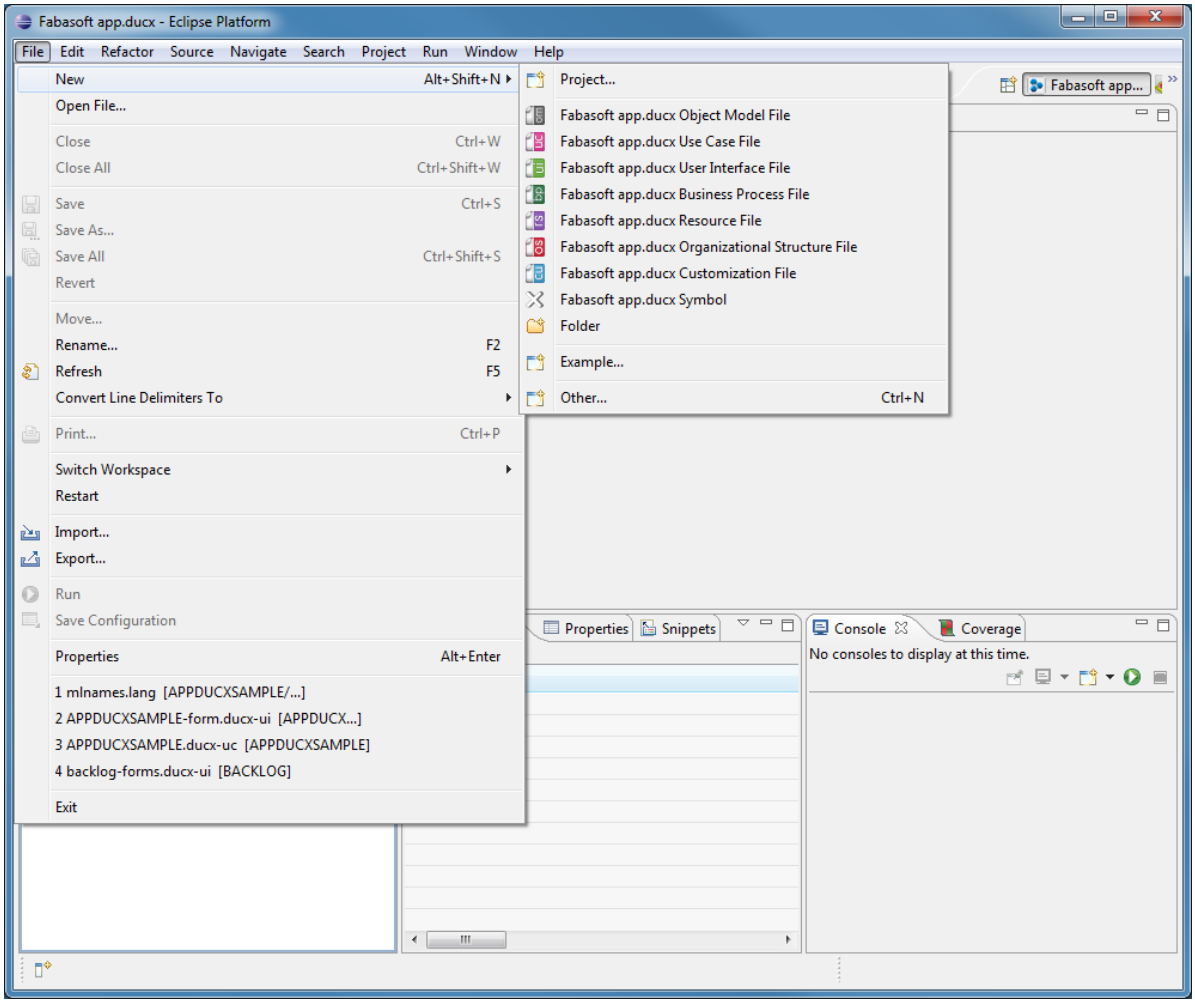


Figure 12: Adding a new source file to a Fabasoft app.ducx project

3.3.4 Adding resources

You can add resources – such as symbols – to a Fabasoft app.ducx project. It is recommended to add all resources to the `resources` folder. Symbols should be added to the `symbols` subfolder.

In Eclipse, a new symbol can be added to your Fabasoft app.ducx project by clicking *New* on the context menu of the Eclipse Project Explorer.

To add an existing resource, just drag the file into the desired folder of your Fabasoft app.ducx project. You may also select *Import* on the context menu of the target folder in the Eclipse Project Explorer.

3.3.5 Exporting a Fabasoft app.ducx project

Exporting a Fabasoft app.ducx project allows you to distribute your project to customers. A Fabasoft app.ducx project represents a single software component. Related software components can be encapsulated in a software product. Several software products build up a software edition. A software edition is a complete product like Fabasoft Folio. To customize a software edition depending on customer wishes is done with a software solution.

Software products, software editions and software solutions are defined as instances within a Fabasoft app.ducx project. For more information how to define such instances see 5.8 “Software products, software editions and software solutions”.

3.3.5.1 Exporting software components

In Eclipse, open the “File” menu and click “Export”. Navigate to “Fabasoft app.ducx” > “Extract Component” to export your Fabasoft app.ducx project to a Fabasoft Folio container file (with a `.coo` extension) that can be installed in another Fabasoft Folio Domain by loading it using the Fabasoft Folio Server Management tool.

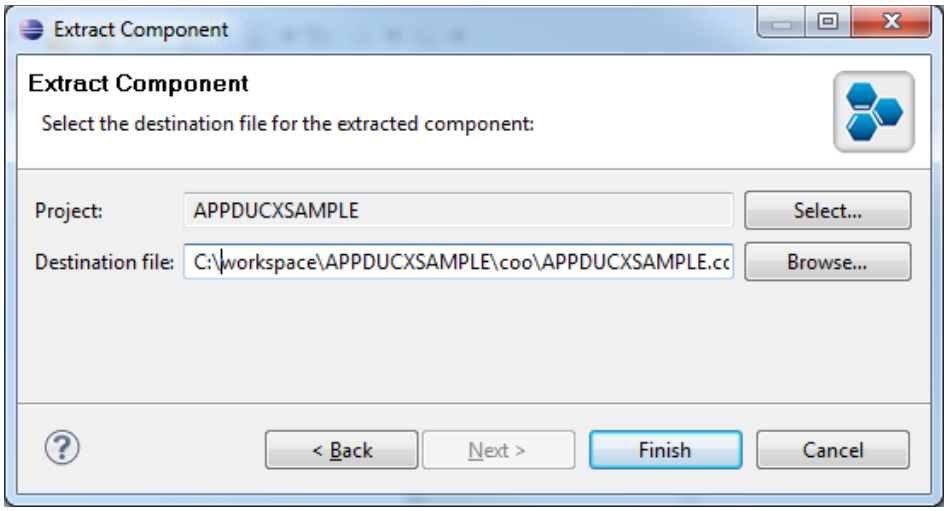


Figure 13: Exporting a Fabasoft app.ducx project

3.3.5.2 Exporting software products, software editions and software solutions

In Eclipse, open the “File” menu and click “Export”. Navigate to “Fabasoft app.ducx” > “Extract Solution” to export a software product, software edition or software solution of your Fabasoft app.ducx project.

3.3.6 Managing address ranges

The address range file and the defined address ranges are provided in the Fabasoft app.ducx project settings. An address range file can be added or changed and additional address ranges can be defined.

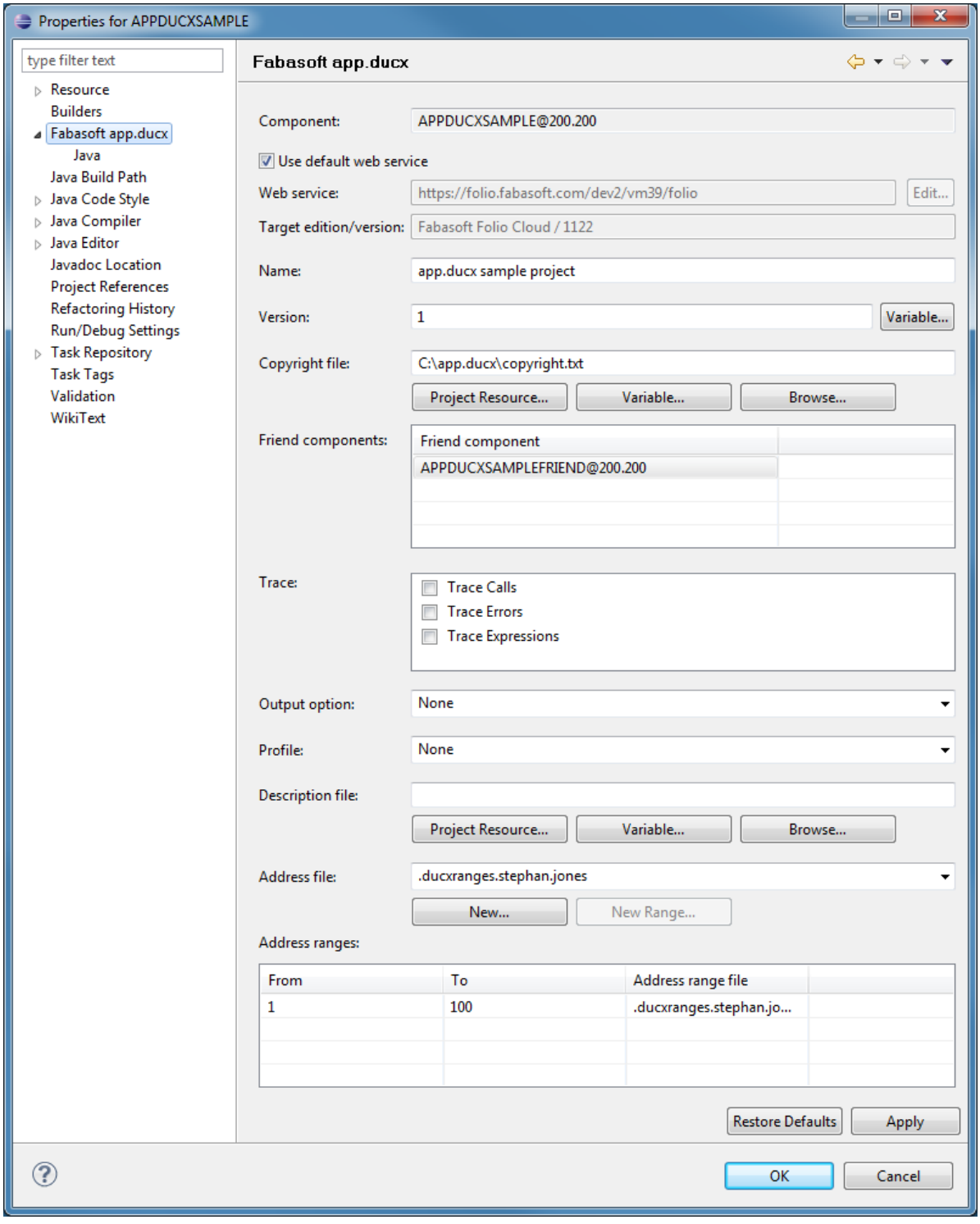


Figure 14: Managing address ranges

Attention:

- Make sure that every developer who works on the same project uses only the address range file that belongs to the developer.
- If a new developer starts working on a project, a new address range file has to be defined for the developer.

In some rare cases it may be useful to change the automatically assigned object addresses. For this purpose an object address editor is available. It can be accessed via the “Address Assignment” entry of a project in the Eclipse Project Explorer. We recommend changing object addresses only if it is absolutely necessary (e.g. if two developer have accidently used the same address range).

3.3.7 Defining a friend component

In the Fabasoft app.ducx project settings a list of friend component references can be provided for your component. The friend components will have access to all the private entities of your component.

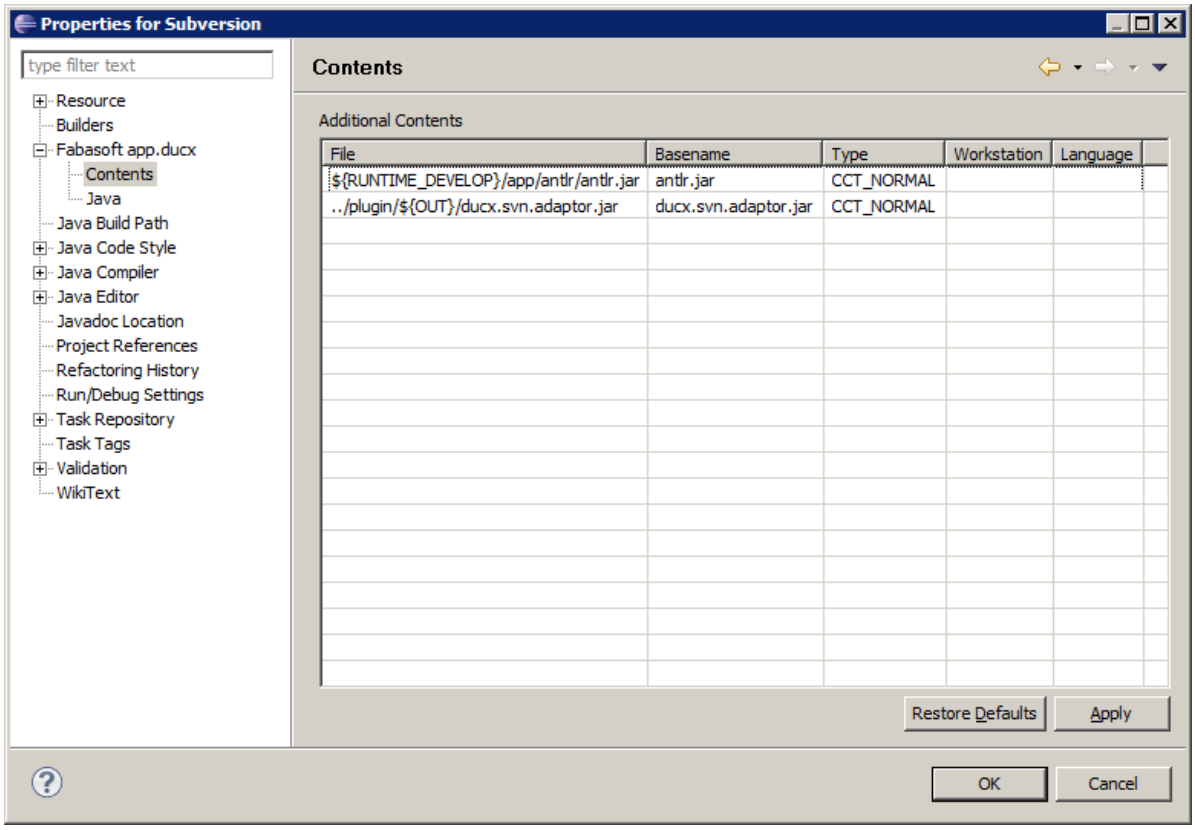
Steps to make a component `APPDUCXSAMPLEFRIEND@200.200` friend of a component `APPDUCXSAMPLE@200.200`:

- In the Fabasoft app.ducx project settings `of APPDUCXSAMPLE@200.200` add `APPDUCXSAMPLEFRIEND@200.200` to the “Friend Components” list.
- Compile and upload `APPDUCXSAMPLE@200.200`
- In project `APPDUCXSAMPLEFRIEND@200.200` add or update reference `APPDUCXSAMPLE@200.200`.

3.3.8 Adding additional contents

Additional contents may be specified within Fabasoft app.ducx project settings. It is not necessary to specify contents like reference documentation, coverage metadata, used jars or generated jars. It is solely for additional contents, e.g. help contents (Compiled Help Files) or contents for a specific platform, e.g. Windows X64 only. These contents will be part of your Software Component.

Contents are simply added by dropping a bunch of files from the filesystem. Additional properties may be set after drop. Basename is initialy constructed of the basename of the file, Type is `CCT_NORMAL` by default. The basename may be changed.

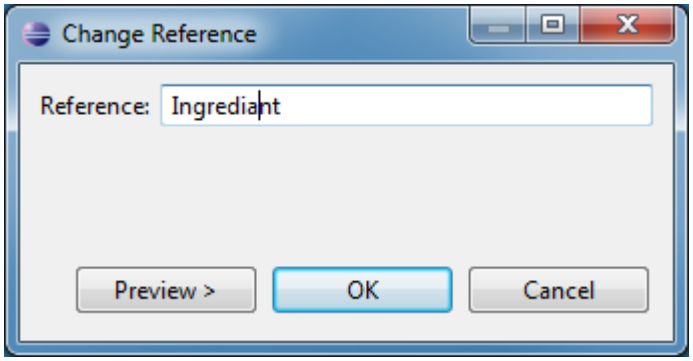


Fabasoft app.ducx tries to identy available variables. The longest match is used. Variables include environment variables and variables available within in Eclipse. However even Eclipse variables may be used, in a build outside of Eclipse, they may not be available.

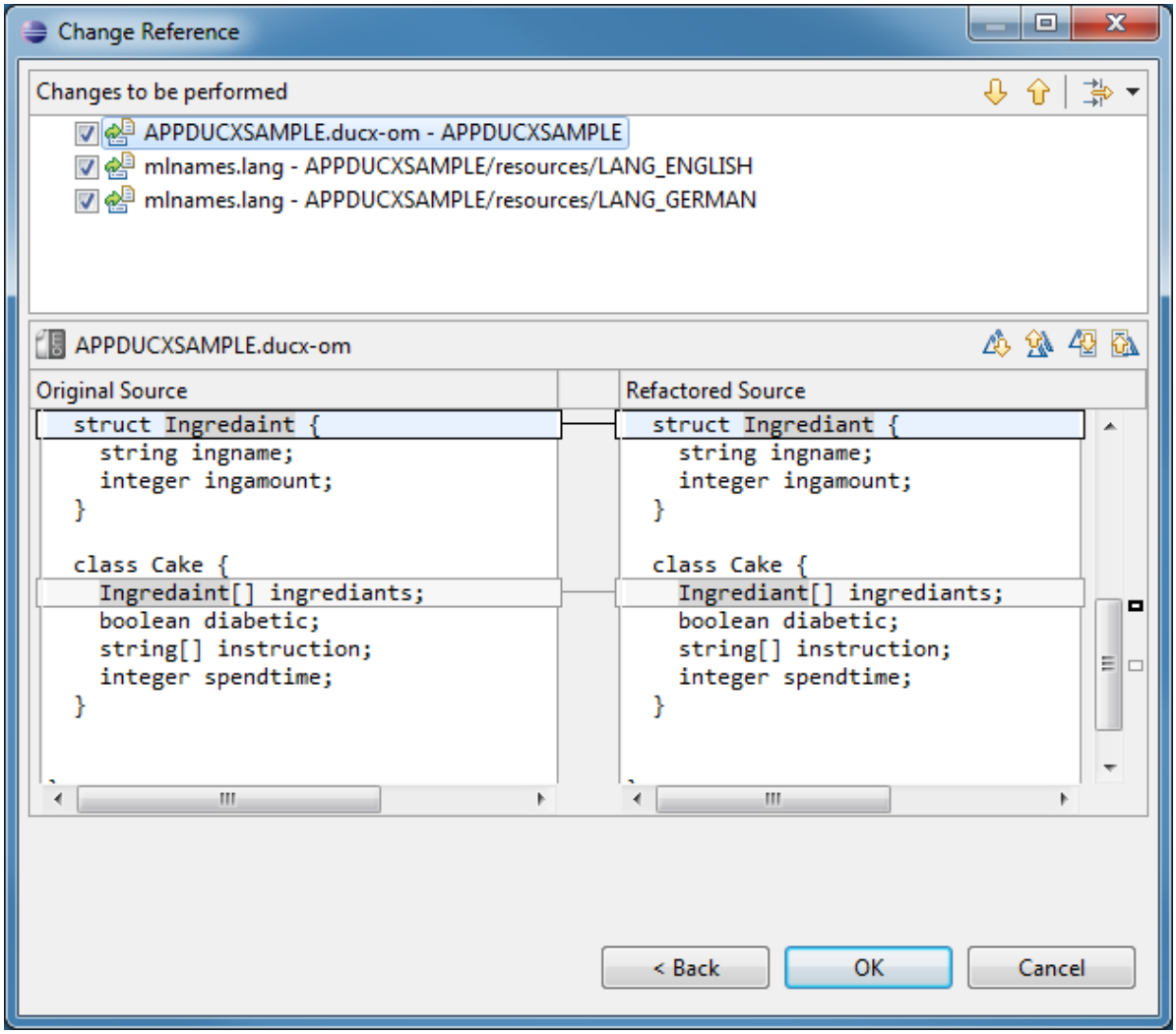
Note: This Page is currently not available for Fabasoft Folio Cloud Apps.

3.3.9 Change references with refactoring

If a reference needs to be renamed, but the address and the multilingual name must not change, refactoring supports this process.



To refactor, select a reference. Open the “Refactor” menu, point to “Fabasoft app.ducx” and click “Change Reference”. In the “Reference” field enter the new reference. Click “Preview” to be able to review the changes or click “OK” to finish the refactoring.



Java and expression source code will not be changed automatically.

3.3.10 Change relation with refactoring

If one of the referenced classes in a relation definition needs to be changed, choose the “Refactor Relation Argument” action in the “Refactor”>”Fabasoft app.ducx” menu. In the “Reference” field enter the reference of the new class, click “Preview” to be able to review the changes or click “OK” to finish the refactoring. The class reference in the relation will be automaticly changed and all the occurences of the implicit attributes of the relation (excepting in expressions) will be renamed according to the changes.

For further information on relation definitions, refer to chapter 5.6 “Defining relations”.

3.3.11 Working together/using a version control system

For Fabasoft app.ducx projects it is also possible to use a version control system like Subversion very easily.

Following files and directories of a Fabasoft app.ducx project have to be checked in:

- `.ranges` (folder)
  Note: All files except the `.mergedducxranges.xml` are required for building your projects.

- `.references` (folder)
  Note: Only `*.xml` (cache, solutions, languages) are required for building your projects. However `*.refdoc` may be checked in for convenience.
- `.references/lib` (folder)
  Note: Only those JARs (e.g. AXIS2 runtime JARs) have to be checked in that have references in the `.classpath` file.
- `.settings` (folder)
  Note: The `.settings` folder contains also developer specific settings for the project (`com.fabasoft.ducx.prefs`). Do not check in or share `com.fabasoft.ducx.prefs` but keep it in order to not lose personal settings like the name of the personal address range file.
- `resources` (folder)
- `src` (folder)
- `.classpath`
- `.ducxproject`
- `.project`
- `*.ducx-*`

Note: Not in every project all listed files and directories exist. Some older projects have instead of the .ranges folder only .ducxranges.* files. These can be replaced with the range files of the .ranges folder.

Several developers may work on a single project. In a project the address ranges of each developer are stored in an own file. During development only the address range file that belongs to the developer gets changed. It is essential that beside the source files also the address range file is checked in.

Generally in Java projects version control files like `.svn` should be excluded from the build process. Open the properties of the project and navigate to “Java Build Path”. On the “Source” tab edit the “Excluded” entry and enter e.g. following line `**/.svn;**/.svn/*` as exclusion pattern.

3.3.12 Customize font and colors

App.ducx provides for the multilingual strings and address assignment editor customizable font and background color properties.

The font and the background color can be defined in the Eclipse preferences (“Window” > “Preferences” > “General” > “Appearance” > “Colors and Fonts”> “app.ducx”).

The font of the app.ducx language editors can be customized by changing the value of “Text Font” in the Eclipse preferences (“Window” > “Preferences” > “General” > “Appearance” > “Colors and Fonts”> “Basic”).

3.4 Updating Fabasoft app.ducx projects using Eclipse

If existing projects should be compiled for a newer version of Fabasoft Folio carry out following steps.

- Make sure that the Fabasoft Folio Domain for development and the Fabasoft app.ducx plug-in have been updated (see chapter 2.4.3 “Updating the Fabasoft app.ducx feature”).
- In the “Project Explorer” navigate to the desired project.
- Open the context menu of the “Software Component References” folder and click “Update All References”.
- Open the “Project” menu and click “Clean”.

This way you get an overview of all warnings and errors in your project that may arise because of changes in the Fabasoft Folio Domain caused by the update. For more information about renamed, deleted and obsolete component objects consult [Faba10e].

3.5 Build and test environments

In build and test environments it is necessary that Fabasoft app.ducx projects can be compiled and loaded using the command line instead of utilizing the Eclipse environment.

To ensure the quality of Fabasoft app.ducx projects an automatic test environment is recommended. Fabasoft app.ducx supports automatic compiling and loading of Fabasoft app.ducx projects in a Fabasoft Folio Domain utilizing Apache Ant. Thus appropriate tests can be carried out immediately after a successful build.

Additionally software products, software editions and software solutions can be extracted. The following chapter describes how to use Fabasoft app.ducx Ant tasks.

3.5.1 Prerequisites

Make sure that Apache Ant is properly installed. For more information about the installation and configuration of Apache Ant consult [ApAn09].

If problems with the Java heap size arise, the values for the Java heap size have to be adjusted accordingly (e.g. `-Xms64m -Xmx1024m`). This can be done by setting the `ANT_OPTS` variable.

- `set ANT_OPTS=-Xms64m -Xmx1024m` (Microsoft Windows)
- `export ANT_OPTS="-Xms64m -Xmx1024m"` or
  `setenv ANT_OPTS "-Xms64m -Xmx1024m"` (Linux; depending on the shell)

3.5.2 Environment variables

The environment variable `DUCX_HOME` is used to specify the path to the needed libraries for compiling and loading. The libraries can be found on the Fabasoft app.ducx DVD in the `Setup\ComponentsAppducx\Ant` directory. At least this environment variable must be set. For example, if you copy the `Ant` directory in the `C:\appducxLibraries` directory, set `DUCX_HOME` to `C:\appducxLibraries`.

The following environment variables may be set in the Fabasoft app.ducx project file. If this is the case, make sure that the environment variables are also set in the build environment or that the corresponding values are defined in the Apache Ant `build.xml`.

- Environment variable for the web service configuration (e.g. `DUCX_WEBSERVICECONFIGFILE`)
  Defines the path to a Fabasoft Folio Web Service configuration file (`.ducxws`). This setting is necessary, if component object references should be updated or the project should be loaded in a Fabasoft Folio Domain.
  Note: To create a configuration file, edit the properties of a Fabasoft app.ducx project in Eclipse and uncheck *Use default web service*. Specify the properties of the web service as needed.
- Environment variable for the copyright file (e.g. `DUCX_COPYRIGHT`)
  Defines the path to a copyright file of the software component defined by the Fabasoft app.ducx project.
- Environment variable for the version (e.g. `DUCX_VERSION`)
  Defines the version of the software component defined by the Fabasoft app.ducx project.
- Environment variable for the target version (e.g. `DUCX_TARGET`)
  Defines the version of the Fabasoft Folio Domain the Fabasoft app.ducx project is compiled for.
  Available target versions: `VERSION81`, `VERSION9`, `VERSION916`, `VERSION925`, `VERSION10`,

VERSION1016, VERSION1026, VERSION1036, VERSION1106, VERSION1116, VERSION1127, VERSION1132, VERSION12, VERSION1227 and VERSION123.

If you compile a Fabasoft app.ducx project containing Java source code the manifest of the generated JAR file can be populated with following environment variables:

- DUCX_SPEC_TITLE=SPEC_TITLE (string)
  Defines the title of the specification.
- DUCX_SPEC_VENDOR=SPEC_VENDOR (string)
  Defines the organization that maintains the specification.
- DUCX_SPEC_VERSION=SPEC_VERSION (string)
  Defines the version of the specification.
- DUCX_IMPL_TITLE (string)
  Defines the title of the implementation.
- DUCX_IMPL_URL (string)
  Defines the URL from which the implementation can be downloaded from.
- DUCX_IMPL_VERSION (string)
  Defines the version of the implementation.
- DUCX_IMPL_VENDOR (string)
  Defines the organization that maintains the implementation.
- DUCX_IMPL_SEALED (true|false)
  Should be set true because all classes defined in all package are archived in the same Fabasoft app.ducx JAR file.

3.5.3 Execute Fabasoft app.ducx Ant tasks

To execute Fabasoft app.ducx Ant tasks you can copy and modify the ant-build-template.xml provided on the Fabasoft app.ducx DVD in the directory Setup\ComponentsAppducx\Ant. Copy the file in a directory and rename the file to build.xml. Modify the build.xml file as described in chapter 3.5.4 “Fabasoft app.ducx Ant tasks” and execute Apache Ant (ant).

3.5.4 Fabasoft app.ducx Ant tasks

Fabasoft app.ducx provides following Ant tasks:

- updaterefs
  The software component cache gets updated based on the defined web service in the Fabasoft app.ducx project file.
- clean
  All intermediate files of a previous compilation are deleted.
- cleanlangfiles
  Cleans up multilingual names and explanation texts, which are not referenced in the source code anymore.
- compile
  The defined Fabasoft app.ducx project is compiled.
- load
  The defined Fabasoft app.ducx project is loaded in the Fabasoft Folio Domain based on the defined web service in the Fabasoft app.ducx project file.
- extract
  The defined Fabasoft app.ducx project is extracted in a COO file.

- `extractsolution`
  Software products, software editions and software solutions specified in the defined Fabasoft app.ducx project are extracted.
- `startcoverage`
  Starts an app.ducx Expression coverage session. A coverage session will live as long until it is explicitly stopped. Sessions not stopped at service shutdown will be persisted and restarted at service startup.
- `stopcoverage`
  Stops an app.ducx Expression coverage session and saves coverage data. For every software component, a coverage data file is provided.
- `ducxunittest`
  Executes the unit tests defined in a given set of components.

To following XML fragment shows the main part of the `build.xml` file. For each project to be compiled define a `ducx` tag. Alternatively you can define one `ducx` tag with a base folder containing several projects to be compiled. Make sure that the copyright, version and target is specified within the `build.xml` file or within the `.ducxproject` file.

Core Example

```
<target name="definewebsvc">
  <webservice id="websvc" url="your baseurl" timeout="your timeout">
    <authentication>
      <basic user="your username" password="your password"/>
    </authentication>
  </webservice>
</target>
<target name="main" depends="declare, definewebsvc">
  <!-- define a ducx tag for each software component to compile -->
  <ducx ducxproject="d:/projects/Symb/.ducxproject" verbose="false">
    <!-- use the web service with ID "websvc" -->
    <webservice refid="websvc"/>
    <!-- define the tasks to be carried out -->
    <!-- clean up multilingual names and explanation texts -->
    <cleanlangfiles copyright="d:/projects/static/copyright.txt"/>
    <!-- delete intermediate files -->
    <clean/>
    <!-- update cache -->
    <updaterefs/>
    <!-- compile the app.ducx project -->
    <compile
      copyright="d:/projects/static/copyright.txt"
      version="1036"
      target="VERSION1036"
      coolib="${env.DUCX_HOME}/Ant/ducx/coolibj.jar"/>
    <!-- load the app.ducx project in a Fabasoft Folio Domain -->
    <load/>
    <!-- extract the app.ducx project to the specified file -->
    <extract target="runtime/symb.coo"/>
    <!-- extract the specified software solution to the target directory -->
    <extractsolution target="runtime" solution="SolutionSymb@200.200"/>
  </ducx>
  <!-- execute only some of the available tasks -->
  <ducx ducxproject="d:/projects/Jmb/.ducxproject" verbose="true">
    <clean/>
    <compile coolib="${env.DUCX_HOME}/Ant/ducx/coolibj.jar"/>
  </ducx>
  <!-- compile all projects in the defined root directory -->
  <ducx root="d:/projs" ducxproject=".ducxproject">
    <clean/>
    <compile coolib="${env.DUCX_HOME}/Ant/ducx/coolibj.jar"/>
  </ducx>
</target>
```

Single File Mode Example

```
<target name="main" depends="declare">
  <!-- define a ducx tag for each software component to compile -->
  <ducx ducxproject="d:/projects/Symb/.ducxproject" verbose="false" single="true">
    <!-- update cache -->
    <updaterefs/>
    <!-- delete intermediate files -->
    <clean/>
    <!-- compile the app.ducx project -->
    <compile
      copyright="d:/projects/static/copyright.txt"
      version="1022"
      target="VERSION1022"
      coolib="${env.DUCX_HOME}/Ant/ducx/coolibj.jar"/>
</target>
```

Coverage Example

```
<target name="coverage" depends="declare, coverage-start, coverage-play, coverage-stop"/>
<target name="coverage-start">
  <startcoverage session="compliance-session-id" verbose="false">
    <!-- session defines a variable name to write the session id into -->
    <!-- webservice defines connection -->
    <webservice url="your baseurl" timeout="your timeout">
      <authentication>
        <basic user="your user" password="your password"/>
      </authentication>
    </webservice>
    <!-- By defining an Edition all assigned components
         (per Software Component Object or as textual reference)
         will be included in measuring -->
    <cover>your Edition</cover>
      <!-- There can be multiple Entries -->
    <cover>your component</cover>
  </startcoverage>
</target>
<target name="coverage-play">
  <!-- do whatever to produce events to be measured by coverage,
       e.g. launch app.test, evaluate unit tests -->
</target>
<target name="coverage-stop">
  <!-- session is a variable holding a session id from a previous startcoverage task
       target is a directory, where coverage information will be placed in
       split defines the breakdown of the different coverage data into separate files
   -->
  <stopcoverage session="${compliance-session-id}" target="coverage" verbose="false"
   split="true"/>
</target>
```

Unit Test Example

```
<target name="main" depends="declare, evaluate-unittests">
<target name="evaluate-unittests" depends="declare">
  <ducxunittest todir="./ducxunitresultdir">
    <!-- enables trace output matching flags and Java Regular Expression -->
    <trace enabled="true" flags="normal, warning, error" clienttimeout="100"
        servertimeout="1000">
      <include>.+unit test.+</include>
    </trace>
    <webservice refid="websvc"/>
    <!-- by defining an Edition all unit tests from the assigned components
         will be executed -->
```

```
    <evaluate>your edition</evaluate>
    <!-- there can be multiple entries -->
    <evaluate>your component</evaluate>
    <evaluate>your unit test group</evaluate>
    <evaluate>your unit test</evaluate>
  </ducxunittest>
  <!-- generate a report e.g. JUnit report -->
</target>
```

3.5.5 Adding dynamic contents to the software component

When a software component is built in the single build mode, additional contents for the software components can be added dynamically without defining them direct in the software component.

To do this, the contents have to be copied in a predefined folder structure in the output folder `build`.

Following rules are applied in the build process:

- Files ending with `".refdoc"` in the help folder get `CCT_DEVHELP` as `COOSYSTEM@1.1:compconttype` and the prefix `"help:"`
- Files in a foldername OperatingSystem get the `COOSYSTEM@1.1:compcontwstype` WSTYPE_OperatingSystem, if the type is available in the enumeration `COOSYSTEM@1.1:WorkStationType`. (e.g. `WINDOWS_X64` gets `WSTYPE_WINDOWS_X64`) and get `CCT_NORMAL` as `COOSYSTEM@1.1:compconttype`.

Example

```
...
<target name="main" depends="declare, definewebsvc">
  <property name="project_dir" value="d:/projects/Symb/" />
  <copy file="${source_dir}/winx64.dll" todir="${project_dir}/build/WINDOWS_X64" />
  <copy file="${source_dir}/centos64.so" todir="${project_dir}/build/CENTOS_X64" />
  <copy file="${source_dir}/helpcontent.so" todir="${project_dir}/build/help" />
  <ducx ducxproject="d:/projects/Symb/.ducxproject" verbose="false" single="true">
    <!-- compile the app.ducx project -->
    <compile
      copyright="${project_dir}/copyright.txt"
      version="1227"
      target="VERSION12"
      coolib="${env.DUCX_HOME}/Ant/ducx/coolibj.jar"/>
  </ducx>
</target>
```

3.5.6 Update app.ducx libraries

To update the app.ducx libraries, download the app.ducx image from `http://update.appducx.com/appducx.iso` and mount it or burn the image on a writeable media. Replace the current libraries from the directory and subdirectories as defined in the environment variable `DUCX_HOME` and update the `build.xml` if necessary. Fore more information about the environment variables see chapter 3.5.2 “Environment variables”.

3.6 Creating your own setup kit

To create a setup kit including your own software solution perform following steps:

1. Copy the Fabasoft Folio product DVD on a writeable media.
2. Copy your software solution in the `Setup` directory.

The base Fabasoft Folio product extended by your own software solution can now be installed or updated using `setup.exe` or `setup.sh`.

Note:

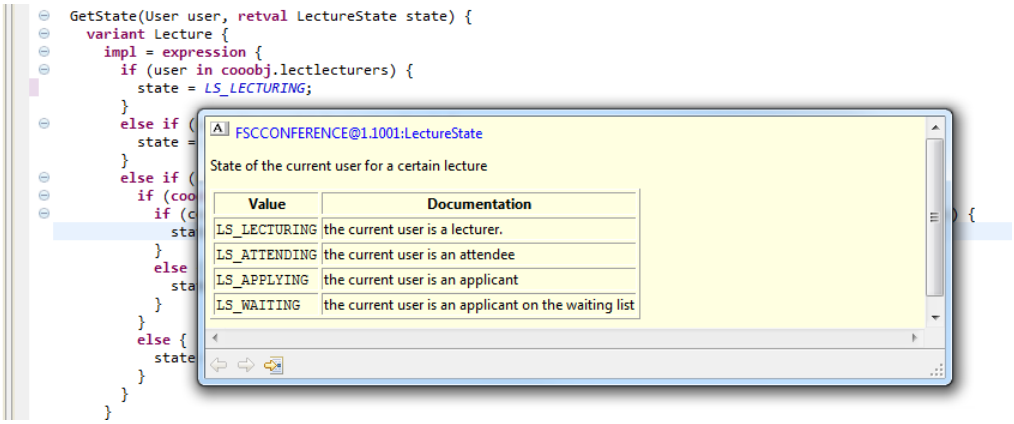
- For more information about software solutions see chapter 5.8 “Software products, software editions and software solutions”.
- The property `COOSYSTEM@1.1:solallowedbaseeditions` of a software solution contains the references of allowed base software editions for this software solution. At least one of the allowed base software editions must be installed in the Fabasoft Folio Domain to be able to install this software solution. Leave this property empty, if the software solution should be a stand-alone solution instead of an add-on solution. Keep in mind that only one stand-alone software edition or software solution can be installed in a Fabasoft Folio Domain.
- Make sure that all COO files of your additionally needed software components (Fabasoft app.ducx projects) are copied in the software solution folder. The software components have to be referenced in the property `COOSYSTEM@1.1:prodcomponents`. Information about exporting software components in COO files can be found in chapter 3.3.5.1 “Exporting software components” and in chapter 3.5 “Build and test environments”.

3.7 Productivity features of Fabasoft app.ducx using Eclipse

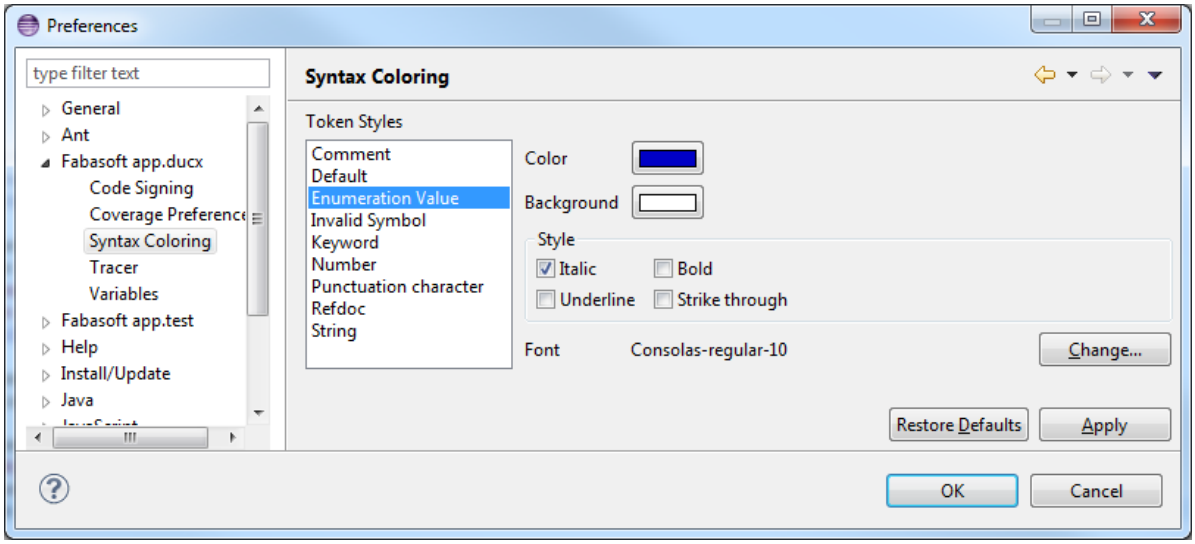
3.7.1 Syntax Highlighting of Enumeration Values

When an enumeration value is used within any Fabasoft app.ducx expression, it is highlighted differently to ordinary identifiers (e.g. local variables). By default, enumeration values are highlighted in blue and italic font, like in the JDT Java editor.

Additionally the reference documentation of the enumeration type is displayed when hovering over an enumeration value.

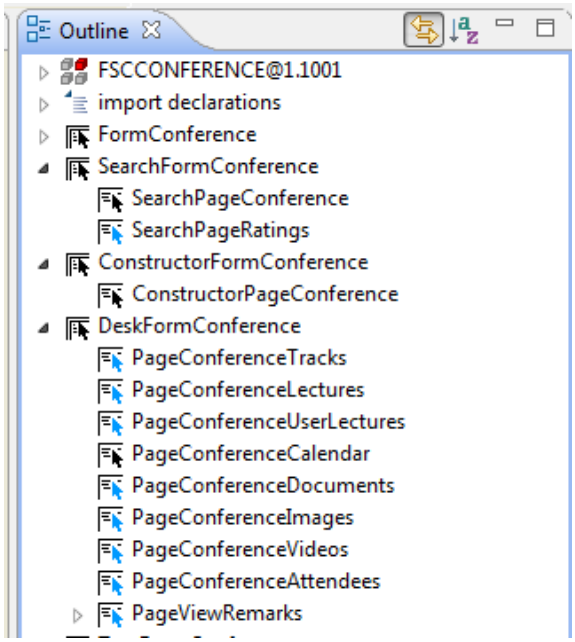


The syntax highlighting can be configured in Windows → Preferences → Fabasoft app.ducx → Syntax Coloring in Eclipse.



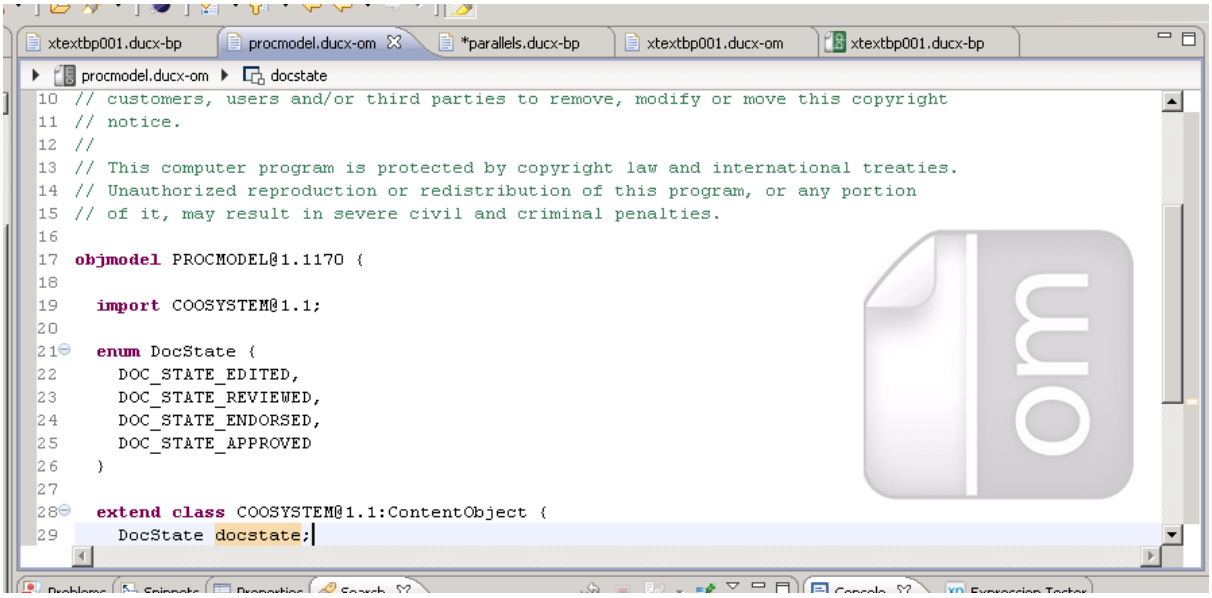
3.7.2 Referenced Form Pages in Outline

Form pages that are not directly defined, but referenced from a form definition are displayed in the outline as well. For convenience, another symbol is used to make it easier to distinguish between defined and referenced form pages.



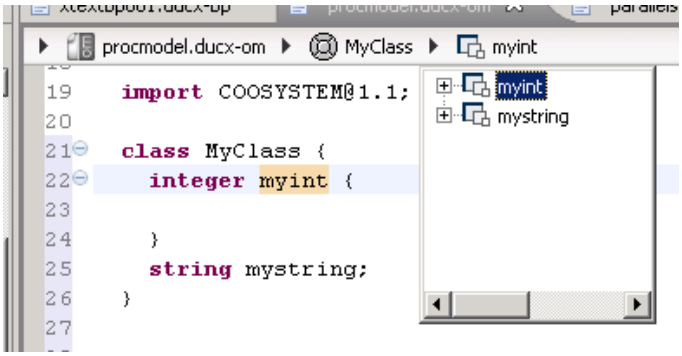
3.7.3 Highlighting of the current domain specific language editor

On the bottom right of the current editor, a symbol of the current domain specific language is shown. This feature requires the Eclipse appearance theme "Classic". This can be set in Windows → Preferences → General → Appearance.



3.7.4 Breadcrumb Navigation

All app.ducx editors support Breadcrumb navigation. The Breadcrumb bases on the current selection or cursor position in the editor.



The Breadcrumb is used to navigate to definition of other objects within the current file or to other files in the same project.

3.7.5 Static check of generic assignments

Generic assignments are checked in a static way. For example, is a property assigned to a class, it will be checked if the property is implemented on the class.

Example

```
objmodel DUCXSAMPLE@200.300
{
  import COOSYSTEM@1.1;
  import COODESK@1.1;
  import FSCVAPP@1.1001;
  class<ContentObjectClass> MyClass {
    classinitcont<contcontent, contextension> = { //OK
      { file("test.txt"), "txt" }
    }
  }
```

```
}
```

```
objmodel DUCXSAMPLE@200.300
{
  import COOSYSTEM@1.1;
  import COODESK@1.1;
  import FSCVAPP@1.1001;
  class<CompoundObjectClass> MyClass {
    classinitcont<contcontent, contextension> = { //NOK: classinitcont is not available on
CompoundObjectClass
      { file("test.txt"), "txt" }
    }
  }
}
```

Data types and assignment are also checked in complex assignments.

The type compatibility of assignment is defined as:

Object class of the attribute	Type of the value
COOSYSTEM@1.1:AttributeIntegerDef	Integer
COOSYSTEM@1.1:AttributeFloatDef	Float
	Integer
COOSYSTEM@1.1:AttributeBoolDef	Boolean
COOSYSTEM@1.1:AttributeDateTimeDef	Datetime
COOSYSTEM@1.1:AttributeEnumDef	String constant oft he enumeration value (equals the definition oft he enumeration type)
	Integer
COOSYSTEM@1.1:AttributeStringDef	String
	Expression
	Integer
	Float
COOSYSTEM@1.1:AttributeExpressionDef	String
	Expression
COOSYSTEM@1.1:AttributeObjectDef	Object
COOSYSTEM@1.1:AttributeEnumDef	Enum
COOSYSTEM@1.1:AttributeContentDef	Content (“file(…)”)

	Expression
	String

3.7.6 Navigation between errors and warnings

Following keystroke combinations are supported:

- "CTRL + ." jumps to the next marker in the current editor
- "CTRL + ," jumps to the previous marker

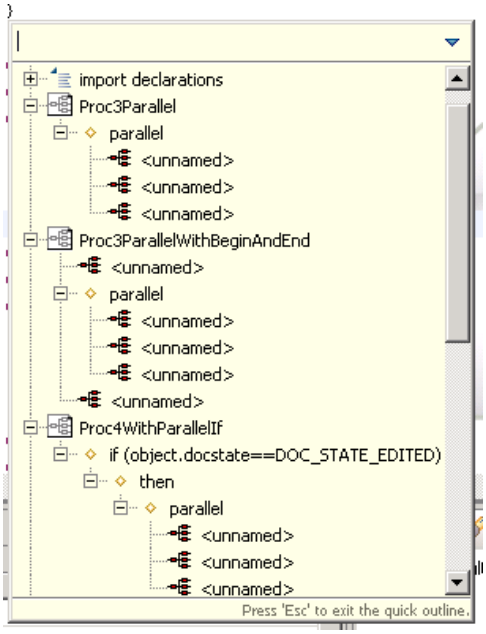
The Eclipse configuration of the markers to jump (errors, warnings, information ...) is in Window > Preferences > General > Editors > Text Editors > Annotations.

3.7.7 Linking between Outline and Editor

The Outline of app.ducx editors is extended with the button "Link with Editor". If this button is checked, the selection adapts itself to the cursor position in the editor and vice versa.

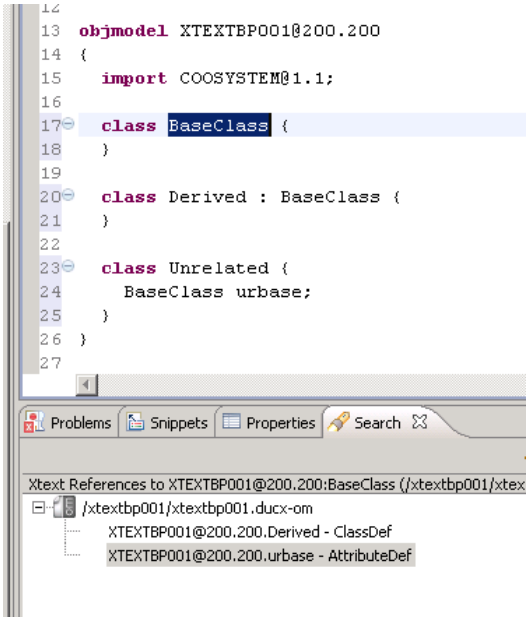
3.7.8 Quick Outline

The keystroke combination "CTRL + O" opens the "Quick Outline" in the current editor. This Outline supports keyboard navigation and contains a quick search for the current source file.



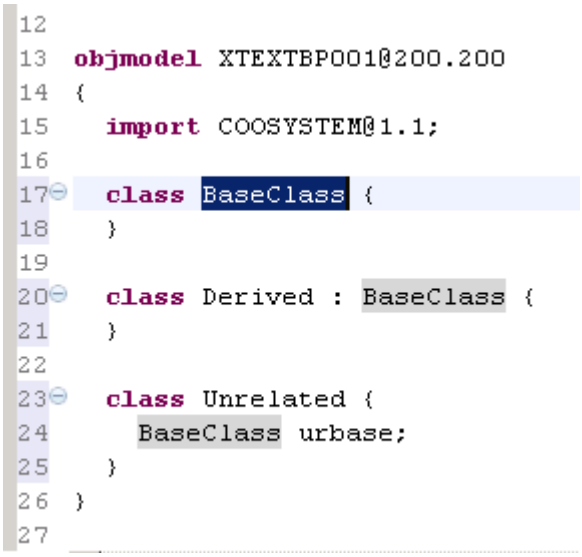
3.7.9 Find References

The keystroke combination "CTRL + Shift + G" is used to find references to the selected object. The search result is displayed in the Eclipse "Search" view.



3.7.10 Mark Occurrences

"Mark Occurrences" is used to highlight references to the currently selected object. This feature is activated via the following button in the menu bar.



3.7.11 Code Formatting

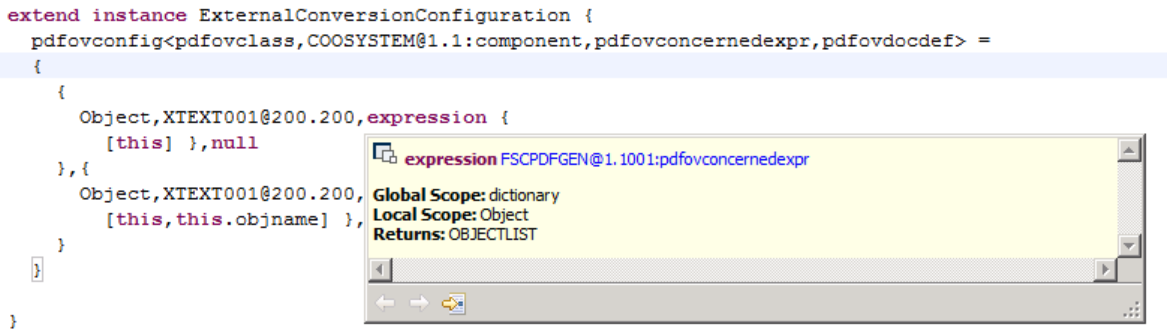
All app.ducx editors support code formatting via `"CTRL + Shift + F"`. This formats indents and line breaks in the current editor.

3.7.12 Color picker in the Content Assist of the UI Editor

In the layout block of form pages, the content assist can be used to assign a color value. After the syntax `fontcolor =` the content assist via `"CTRL + Space"` provides a color picker for the favored color.

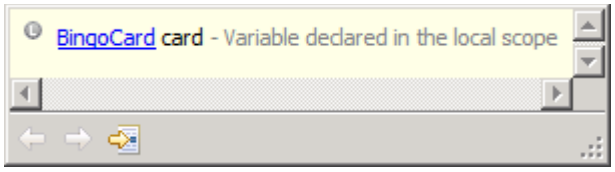
3.7.13 Expression informationen in complex assignments

The information about available scopes and parameters of the current expression is also shown if the mouse pointer is placed on the `expression` keyword. This feature enables the scope and parameter information of expressions in complex assignment.



3.7.14 Information about variables in Expressions

The type and the scope of a variable are displayed in the tooltip of variables in expressions. The type is only displayed if resolvable.



3.7.15 Goto definition for variables in Expressions

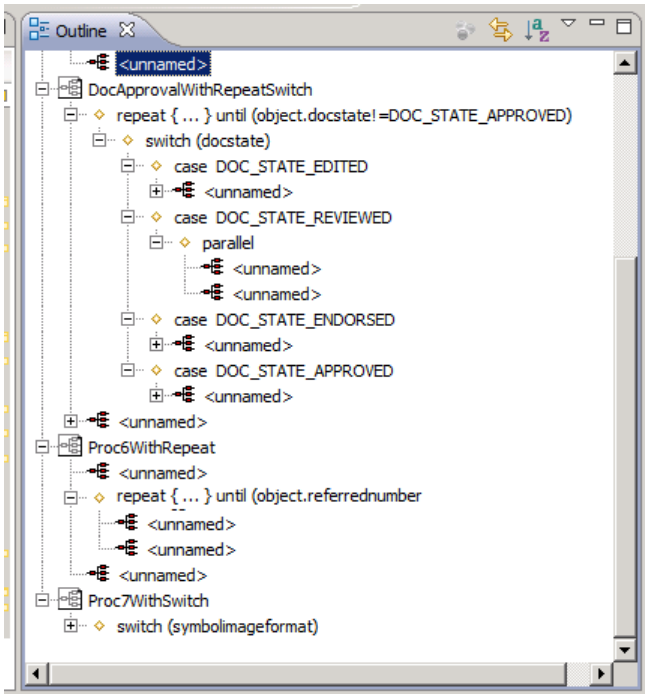
The feature "Goto definition" is available for variables with type declaration in expression. Pressing `F3` on a variable will jump to the variable declaration. If the variable is defined as an action or use case parameter, the cursor jumps to formal parameter declaration.

3.7.16 Copy a reference from the Project Explorer

The Project Explorer is extended with a context menu "Copy Reference" on the elements, which copy the reference of the selected object to the clipboard.

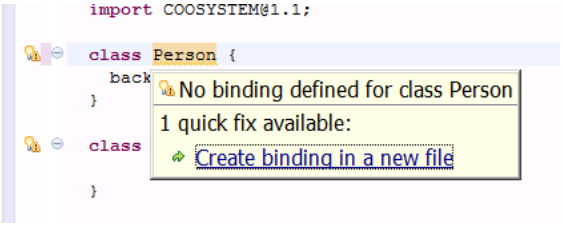
3.7.17 Outline in the Business Process Editor

The Outline of the Business Process Editor is improved with a concrete structure of complex process definitions. Process elements like conditions, loops, parallel or sequential blocks are shown within the process definition.



3.7.18 Warning for missing user interface binding at object class declarations

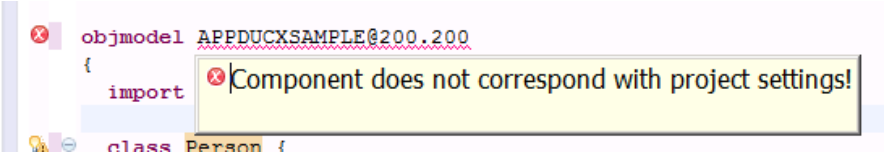
If an object class has no user interface binding, a warning is shown.



Additionally, a Quick Fix is provided to create the binding either in an existing or in a new file.

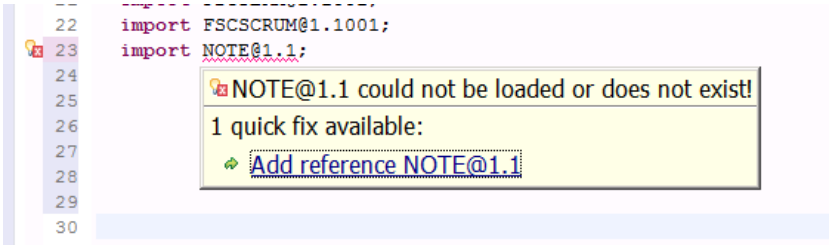
3.7.19 Error for mismatching component and model

If the software component of the source file differs to the software component of the project, an error is shown.

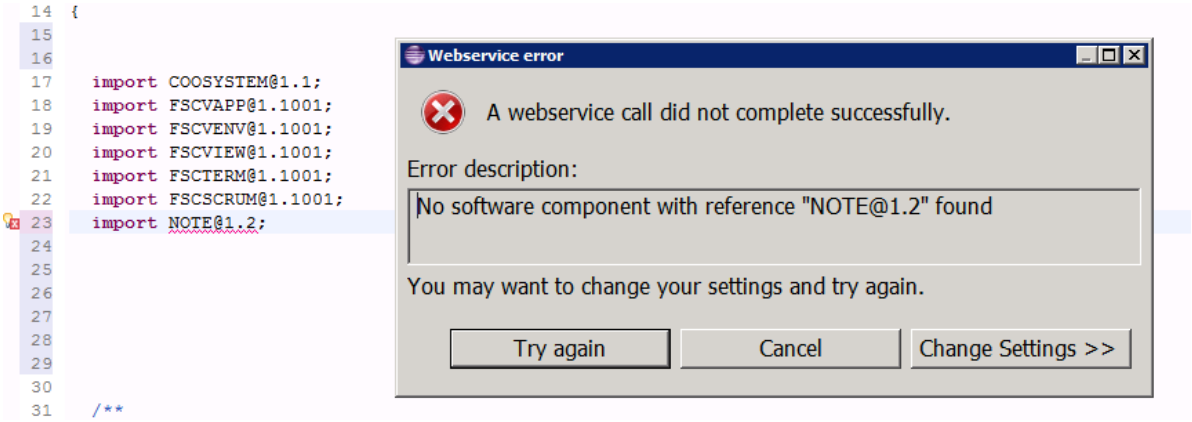


3.7.20 Quick fix to add a reference to the project

If a software component is imported in a source file and it is not in the references of the current project, a Quick Fix to add the software component to the project is provided.



If this software component is not available in the configured web service or the web service itself is not available, an error message is shown.



3.7.21 Content Assist

Fabasoft app.ducx provides Content Assist support within expression blocks and domain specific languages. Content Assist may be triggered manually with `Ctrl` + `Space`. But in several situations Content Assist gets triggered automatically:

- `::`
  accesses the global scope
- `:>`
  accesses the local scope
- `@`
  accesses the temporary scope
- `#`
  retrieves a component object
- `.`
  may be followed by a Fabasoft Folio Kernel Interface Method, a property, an action or a use case

3.7.22 Quick Fix for ambigious elements

If short references are ambiguous within the Fabasoft app.ducx project, a “Quick Fix” is provided. You can choose between possible fully qualified references and replace the short reference with the desired fully qualified reference.

If an unknown identifier has been found during the code validation and a corresponding app.ducx construct could be determinated app.ducx offers a quick fix to create such an entity.

3.7.23 Folding

Source code blocks starting with defined keywords like `class` or `enum` can be folded to get a better overview of the source code.

3.7.24 Automatically close brackets

After typing `{`, `[`, `(` in the source code the corresponding .close bracket is automatically created.

3.7.25 Find the matching bracket

If the cursor is behind an opening or closing bracket its counterpart is highlighted. This option can be enabled or disabled on the “Fabasoft app.ducx” > ”Preferences” > ”Editor” preference page.

To jump to the counter part of a bracket, put the cursor behind the opening or closing bracket and press `Ctrl + Shift + P`. The shortcut key options can be changed in the Eclipse preferences (“Window” > “Preferences” > “General” > “Keys” > “Go to matching bracket”).

3.7.26 Tooltips

If you point to a short reference, the corresponding fully qualified reference is displayed as tooltip. A hyperlink allows viewing the properties of the component object with the Fabasoft Folio Web Client.

3.7.27 Edit component object instances utilizing the Fabasoft Folio Web Client

Instances of component objects like

- *ACL* (`COOSYSTEM@1.1:AccessControlList`),
- *Data Import (Component Object)* (`FSCCOLD@1.1001:DataImportComponentObject`),
- *Active Report (Text)* (`COOAR@1.1:ActiveReport_Text`),
- *Relational Report Definition (Component Object)* (`FSCRELREP@1.1001:RelRepDefComponentObject`) and
- *Web Service Definition* (`FSCOWS@1.1001:WebServiceDefinition`)

can be edited utilizing the Fabasoft Folio Web Client. The instance gets transformed to Fabasoft app.ducx source code that is inserted in your project. Thus complex compound properties can be edited very efficiently using a graphical user interface.

In the project explorer navigate to the desired instance and execute the context menu command “Edit”. The default web browser is opened and the instance can be edited. Click ”Next” to transform the instance to Fabasoft app.ducx source code or “Cancel” to discard the changes. The context menu command “Synchronize” can be used to transform the current instance of the Fabasoft Folio Domain to Fabasoft app.ducx source code.

Note:

- While editing the component object in the Fabasoft Folio Web Client no editing in Eclipse is possible.
- Only changes of the opened instance are considered. Changes of objects that are referenced in object pointer properties of the opened instance are not transformed to Fabasoft app.ducx source code.

3.7.28 Templates

The former used code snippets for inserting predefined pieces of code are now provided by templates via the Content Assist.

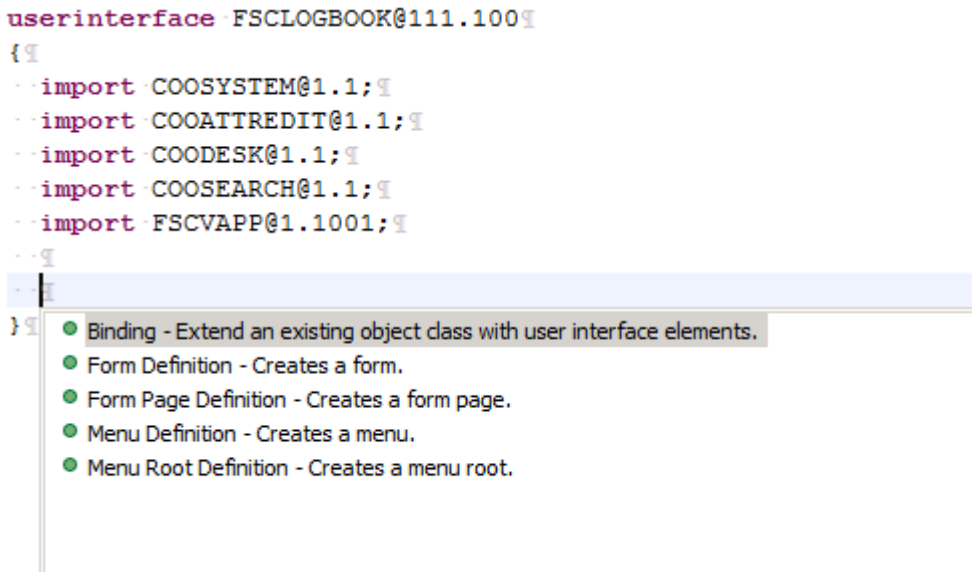


Figure 15: Using code templates

Templates are marked in the Content Assist with a green dot.

3.7.29 Spell checker

For comments a spell checker is provided. The dictionary can be defined in the Eclipse preferences ("Window" > "Preferences" > "General" > "Editors" > "Text Editors" > "Spelling"). Additionally a user defined dictionary can be provided to be able to add unknown words to this dictionary.

Right-click on a misspelled word denoted by a red underline and click "Quick Fix" (shortcut: `Ctrl` + `1`). You can select a suggestion, add the word to the user dictionary, ignore the word during the current session or disable spell checking.

3.7.30 Fabasoft Reference Documentation

The object model of Fabasoft products is subsumed in the Fabasoft Reference Documentation http://help.appducx.com/?topic=doc/Reference-Documentation/index.htm.

The Fabasoft Reference Documentation is also displayed context-sensitively as tooltip in the Eclipse environment.

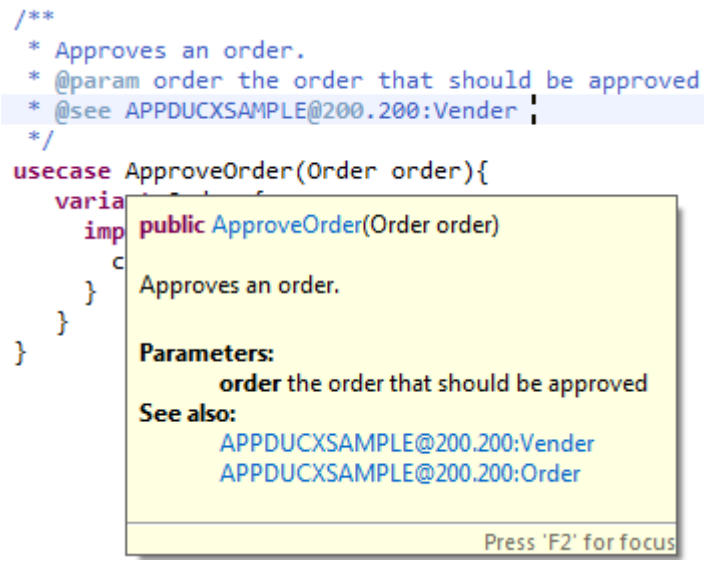


Figure 16: Customizing snippet code

The manual description is taken from source code comments e.g. for enumeration types, object classes, use cases or properties. When typing `/**` and pressing `Enter` e.g. parameters of a use case are generated automatically for easier documentation.

If a component reference should be displayed in the “See also” block, the full reference has to be used after the `@see` tag. Useful references (e.g. type of the parameters, members of an object class…) are generated implicitly.

A description for a software component itself may be defined in the properties of a project (“Properties” > “Fabasoft app.ducx” > “Description file”). The text file for the description may contain XHTML tags for structuring the text (e.g. `<p></p>`, `<ul></ul>`, `<br/>`).

Note: For a complete description of the syntax for writing reference documentation comments refer to [Orac10b].

4 Domain-Specific Language Fundamentals

This chapter gives an introduction to the concept of domain-specific languages (DSLs) as well as a concise summary of important recommendations to keep in mind when working with Fabasoft app.ducx.

4.1 What is a DSL?

Developing use case-oriented software solutions requires managing different aspects and elements such as data structures, user interface design, the implementation of methods and business rules.

In order to account for this concept in an optimal manner, Fabasoft app.ducx is comprised of several declarative modeling languages, each designed for covering a particular aspect of solution development. For example, Fabasoft app.ducx includes a modeling language that has been designed explicitly for the definition of an object model. In addition to this, Fabasoft app.ducx includes languages for defining resources, a user interface model, an implementation model, a process model, and an organizational structure model.

These modeling languages are referred to as domain-specific languages (DSLs), where each DSL was designed for addressing a certain aspect of use case-oriented software development. The modular concept makes Fabasoft app.ducx easily extensible as new DSLs can be added on demand for addressing additional aspects without affecting existing projects.

Currently, Fabasoft app.ducx is comprised of eight distinct DSLs:

- app.ducx Object Model Language
- app.ducx Resource Language
- app.ducx Use Case Language
- app.ducx Business Process Language
- app.ducx User Interface Language
- app.ducx Organizational Structure Language
- app.ducx Customization Language
- app.ducx Expression Language

Each of these DSLs is covered in detail in its own chapter (except for the app.ducx Customization Language, which is beyond the scope of this book).

4.2 Common characteristics of DSLs

Each DSL is an independent declarative modeling language having its own set of language constructs and keywords designed with a particular purpose in mind. However, all DSLs also share certain similarities. These common characteristics and language elements that are the same across all DSLs of Fabasoft app.ducx are presented in this section.

4.2.1 Keywords, expressions and blocks

A DSL consists of a unique set of *keywords* for addressing a particular aspect of solution development. However, there are shared keywords that are common to all DSLs.

The Fabasoft app.ducx compiler is case-sensitive. All keywords must be in lower case.

Each expression must be terminated by a semicolon or be enclosed in a block. A block consists of one or more expressions enclosed in curly braces.

4.2.2 Operators and punctuators

There are several kinds of operators and punctuators. Operators are used in expressions to describe operations involving one or more operands.

Examples for operators and punctuators are `{ } [ ] ( ) , . : ; + - * / % & | ^ ! ~ = < > ? ++ -- && || == != <= >= <=> += -= *= /= ->`.

4.2.3 Comments

Two forms of comments are supported: single-line comments and delimited comments. Single-line comments start with the characters `//` and extend to the end of the source line. Delimited comments start with the characters `/*` and end with the characters `*/`. Delimited comments may span multiple lines. Comments do not nest and are not processed within character and string literals.

4.2.4 Multilingual strings

Multilingual strings are not defined in the source code but in separate resource files in order to keep the source code free of any hard-coded strings. By default, the resource files holding multilingual strings are named `mlnames.lang`. For each language you want to support, a separate resource file must be created.

The main benefit of this approach is that all language-specific string resources are maintained in a single resource file for the particular language, which allows for a quick and hassle-free localization of your software application.

Resource files are automatically updated whenever you add a new element requiring a multilingual name (e.g. object classes, properties, or use cases) or another form of multilingual string, such as the label text for an input field on a form page or the text displayed for enumeration items in a drop-down list.

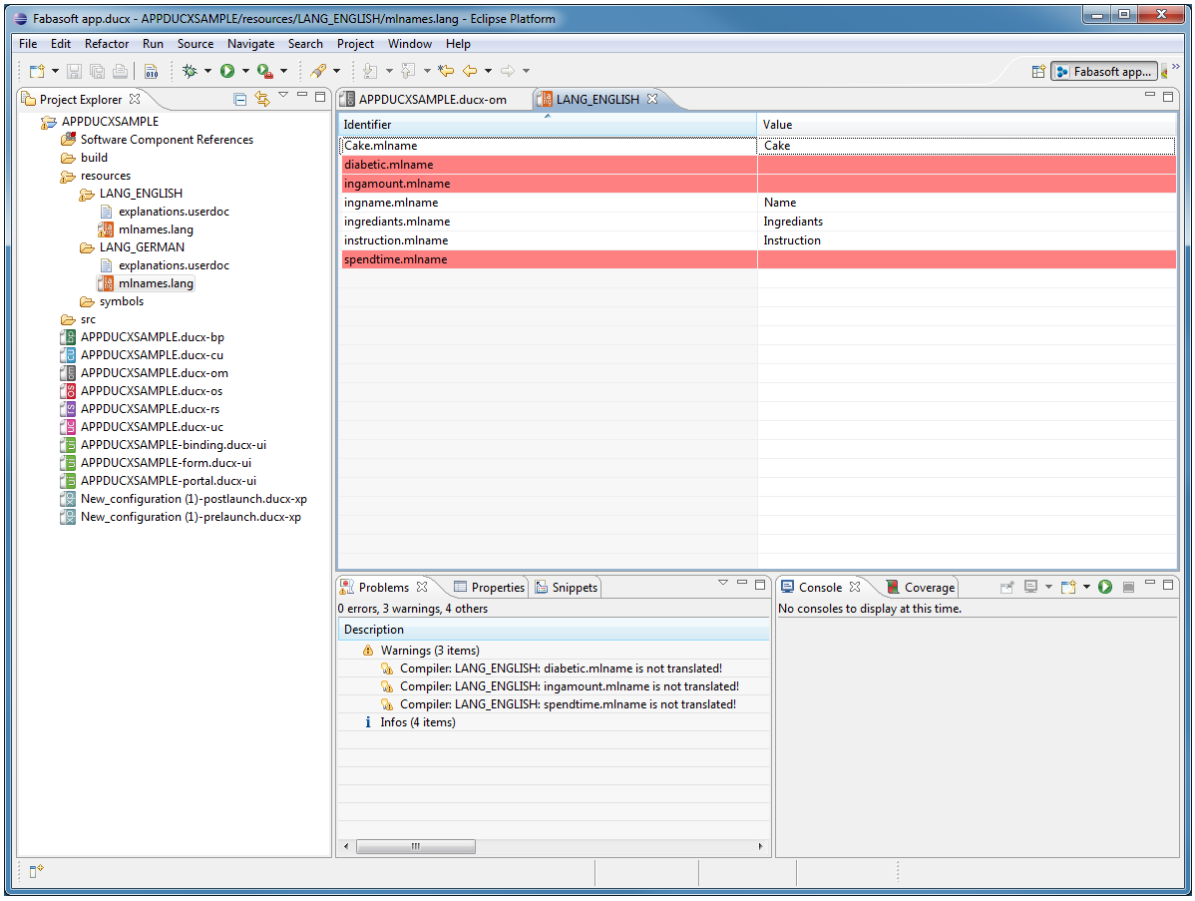


Figure 17: Defining multilingual strings

If possible, each multilingual string will be initialized by Fabasoft app.ducx with the reference supplied for the corresponding element. However, you can edit the multilingual strings for all elements that are part of your software component by opening the `mlnames.lang` file for the language you want to edit in the Eclipse Project Explorer.

The language resource editor (see previous figure) is invoked when opening the `mlnames.lang` file. It allows you to enter a language-specific string for each multilingual string resource in your app.ducx project.

If a language string is not yet translated, the background color of this line is changed and after a successful build of this project, a warning is generated.

Beside `mlnames.lang` files, `explanations.userdoc` files (XML) are provided for defining explanation texts for properties. Simple XHTML tags like `<b></b>`, `<i></i>` or `<br/>` may be used to format the strings. These explanation texts can be displayed as context-sensitive help in the Fabasoft Folio Web Client. For detailed information on the help system, please refer to [Faba10d].

Note: For your convenience it is recommended that you use an XML editor for editing explanation texts. To assign the `userdoc` file extension to your favorite XML editor open the “Window” menu and

click “Preferences”. Navigate to “General” > “Editors” > “File Associations”. Add the `userdoc` file extension and associate the desired editor. The editor may be an internal Eclipse plug-in or an external XML editor.

Multilingual strings and explanation texts are not removed automatically when the corresponding reference is removed from the source code. Thus, you do not lose strings when temporarily commenting out some source code. To clean up multilingual names and explanation texts, which are not referenced in the source code anymore, use the context menu command “Clean up Language Files”.

4.2.5 Using fully qualified references for referring to component objects

In all DSLs, component objects can be addressed by their *fully qualified reference*.

Each component object has a unique programming name, which is described by its fully qualified reference that indicates a logical hierarchy. For example, `COOSYSTEM@1.1:objsubject` is the fully qualified reference of a component object with the *reference* `objsubject` that is part of software component `COOSYSTEM@1.1`.

In Fabasoft app.ducx, a software component is somewhat similar to a namespace in an object-oriented programming language: all component objects belonging to a particular software component are nested in this namespace.

The following example shows part of an object model containing the definition of object class `Product`. This object class is derived from object class `COOSYSTEM@1.1:BasicObject`. Furthermore, the existing property `COOSYSTEM@1.1:mlname` is assigned to the object class. Please note that fully qualified references are used for referring to both of these component objects belonging to software component `COOSYSTEM@1.1`.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  class Product : COOSYSTEM@1.1:BasicObject {
    COOSYSTEM@1.1:mlname;
  }
}
```

4.2.6 Using the import statement

Using the import statement, a software component can be made known to the compiler in order to avoid the need for using the fully qualified reference when referring to a component object that is part of the imported software component.

Once a software component has been imported using an import statement, you can refer to component objects belonging to this software component with their reference only, instead of having to use the fully qualified reference. This allows for the *software component prefix* to be left out when referring to its component objects once a software component has been imported.

A list of import statements is referred to as import declarations. All import declarations have to be placed inside `objmodel`, `resources`, `userinterface`, `usecases`, `processes` or `orgmodel` blocks. The different types of models are described in greater detail in the following chapters.

Note: Software components listed in the import declarations must be referenced in your Fabasoft app.ducx project’s list of software component references. For further information on software component references, refer to chapter 3.3.2 “Adding a software component reference”.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
```

```
  // Importing COOSYSTEM@1.1 permits the use of reference shorthands when
  // referring to component objects that belong to COOSYSTEM@1.1, instead
  // of requiring the use of fully qualified references
  import COOSYSTEM@1.1;
  class Product : BasicObject {
    mlname;
  }
}
```

It should be noted that even when a software component is imported using an import statement it is still valid to refer to component objects of the imported software component using the fully qualified reference.

4.2.7 Resolving of not fully qualified references

A reference that is not fully qualified may be ambiguous, because component objects belonging to different software components might have the same reference. In this case, the fully qualified reference must be used to refer to these component objects. However references of the own software component are favored over references of other software components.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCFOLIO@1.1001;
  import FSCREC@1.1001;
  // "Record" is not sufficient for unique qualification, because it could
  // refer to either FSCFOLIO@1.1001:Record or FSCREC@1.1001:Record
  class DUCXRecord : FSCFOLIO@1.1001:Record {
    ...
  }
  class SpecialRecord : FSCREC@1.1001:Record {
    ...
  }
  class State : FSCFOLIO@1.1001:State {
    ...
  }
  // "State" refers to APPDUCXSAMPLE@200.200:State because the own
  // software component is favored
  class SpecialState : State {
    ...
  }
}
```

4.2.8 Resolving of qualifiers in app.ducx expressions

Beyond the resolving strategy of not fully qualified references some more rules apply to app.ducx expressions.

As a general rule of thumb, whenever you use qualifiers provide a type for the qualifiers to avoid ambiguities.

In following cases special resolving rules apply:

Compound properties

1. Attempt to resolve the qualifier to a Fabasoft Folio Kernel Interface Method.
2. Attempt to resolve the qualifier to a property of the compound property.
3. No further resolving attempt is carried out and an error is generated.

In the following example `GetTypeDefinition` is resolved to a Fabasoft Folio Kernel Interface Method and `isbn` to a property of the compound property `publication`.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  usecase ResolveCompoundProperty() {
    variant Order {
      impl = expression {
        Publication @publication = {};
        @publication.GetTypeDefinition();
        @publication.isbn;
      }
    }
  }
}
```

Objects

1. Attempt to resolve the qualifier to a Fabasoft Folio Kernel Interface Method.
2. Attempt to resolve the qualifier to a property of the object.
3. Attempt to resolve the qualifier to an action or use case of the object.
4. No further resolving attempt is carried out and an error is generated.

In the following example `GetClass` is resolved to a Fabasoft Folio Kernel Interface Method, `orderpositions` to a property of the object `order` and `ResolveCompoundProperty` to a use case of the object `order`.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  usecase ResolveObject() {
    variant Object {
      impl = expression {
        Order @order = #Order.ObjectCreate()[2];
        @order.GetClass();
        @order.orderpositions;
        @order.ResolveCompoundProperty();
      }
    }
  }
}
```

Item Scope

1. Attempt to resolve the qualifier according to the rules for the type of the list.
2. Attempt to resolve the qualifier according to the rules of the local scope.
3. No further resolving attempt is carried out and an error is generated.

In the following example `objname` is resolved to `COOSYSTEM@1.1:objname` and `name` to a key in the local scope.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
```

```
  usecase ResolveObjectList(string name) {
    variant Object {
      impl = expression {
        Object[] objects = coort.SearchObjects(cootx, "SELECT * FROM
                 COOSYSTEM@1.1:Object WHERE objname LIKE 'Test%'");
        objects[objname == "Test"];
        objects[objname == name];
      }
    }
  }
}
```

Dictionaries

1. Attempt to resolve the qualifier to a Fabasoft Folio Kernel Interface Method.
2. Attempt to resolve the qualifier to a dictionary key.
3. No further resolving attempt is carried out and an error is generated.

In the following example `key` is resolved as dictionary key.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  usecase ResolveDictionary() {
    variant Object {
      impl = expression {
        dictionary @dict;
        @dict = coort.CreateDictionary();
        @dict.key = "Test";
      }
    }
  }
}
```

Other data types

1. Attempt to resolve the qualifier to a Fabasoft Folio Kernel Interface Method.
2. No further resolving attempt is carried out and an error is generated.

4.2.9 Using generic assignment statements

Fabasoft app.ducx allows you to include so-called generic assignment statements in your source code in order to assign values to Fabasoft Folio properties. You may only reference properties that actually belong to the object class of the component object denoted by the Fabasoft app.ducx language element the assignment statement is nested in.

4.2.9.1 Initialization of properties with scalar values

Syntax

```
property = value;
```

For initializing a property with a scalar value, the reference of the property must be denoted followed by the equality character, and the value. The assignment statement must be terminated by a semicolon.

In the following example, the `COOSYSTEM@1.1:attrextension` property of a content property should be initialized with the value “txt”. However, there is no language element provided by any of the

Fabasoft app.ducx DSLs for defining the value for this property. Thus, a generic assignment statement must be used for defining the value of this property as illustrated by the following example.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCFOLIO@1.1001;
  class OrderRecord : Record {
    content orderspecification {
      attrextension = "txt";
    }
  }
}
```

4.2.9.2 Initialization of compound properties

Syntax

```
property<member, ...> = {
  {value, ...}
}
```

When assigning a list of values to a compound property, a special syntax must be used. The reference of the compound property to be initialized must be followed by angle brackets. In the angle brackets, the references of the properties belonging to the compound property's type that should be initialized with values must be specified. The initialization values provided must be enclosed in curly braces and separated by commas. Null values can be defined using the `null` keyword. Additionally, multiple lines must be separated by commas as well.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCFOLIO@1.1001;
  import COODESK@1.1;
  class OrderRecord : Record {
    classdocstateicons<dsirecorded, dsidocstate, classmicon> = {
      {true, DS_EDIT, MiniIconRecord_Edit},
      {true, DS_SUSPENDED, MiniIconRecord_Suspended},
      {true, DS_CLOSED, MiniIconRecord_Closed},
      {true, DS_CANCELLED, MiniIconRecord_CancelledRec}
    }
  }
}
```

Compound properties can also be nested within other compound properties. The following example demonstrates the initialization of nested compound properties.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCFOLIO@1.1001;
  import COOWF@1.1;
  import COODESK@1.1;
  extend class WorkList {
```

```
    dispviews<dispviewattr, dispviewcomp, dispcolumns<dispattribute>> = {
      {worklistitems, APPDUCXSAMPLE@200.200,
        {
          {objname},
          {actinstwork},
          {{actinstobject, bocontact}},
          {{actinstobject, boassignedto}},
          {actinstremark},
          {actinstreceivedat},
          {actinstenddeadline}
        }
      }
    }
  }
}
```

4.2.10 Public, private and obsolete

Component objects may be marked as public, private or obsolete using the keywords `public`, `private` and `obsolete`.

Following rules apply:

- Component objects are public by default except in following cases:
  - Actions are private by default but can be marked as public.
  - Activities that are defined within the scope of a process are private by default but can be marked as public.
- Implicitly generated component objects not defined in the source code (like menus or applications for a use case) are public, private or obsolete depending on the definition of the component object itself that triggers the implicit generation of component objects.

Cloud profile note: All component objects are by default private. They must be explicitly marked as public when required.

4.2.10.1 Private

Private component objects can only be used within the software component the component object is created in and in friend components (For further information on how to create a friend relation, please consult chapter 3.3.7 “Defining a friend component”).

The type of the parameters of a public action or a public attribute must be also public. The private constraint is enforced by Fabasoft app.ducx the following way:

- When using external private component objects of a non-friend component an error is generated.
- External private component objects in non-friend components are excluded from IntelliSense.
- Java: External private component objects in non friend components are not available.

4.2.10.2 Obsolete

Obsolete component objects denote component objects that should not be used anymore and that are likely no longer available in prospective versions. To determine obsolete component objects Fabasoft app.ducx provides following support:

- When using obsolete component objects a warning is generated.
- Obsolete component objects are excluded from IntelliSense.
- Java: Obsolete component objects are marked as deprecated.

4.2.11 Referencing resources

The `file` keyword is used to reference a resource such as an image or another type of binary or text file. It must be followed by parentheses holding the relative path to the resource file enclosed in double quotes. The path must be specified relative to your Fabasoft app.ducx project.

The `file` keyword can also be used to reference app.ducx expressions that have been defined in separate app.ducx expression language files.

Example

```
// Referencing a symbol imported to the "resources/symbols" folder of the
// app.ducx project
symbol SymbolApprove {
  images<symbolimageformat, content> = {
    {
      SF_PNG16,
      file("resources/symbols/Approve16.png")
    }
  }
}
// Referencing an expression in an app.ducx expression language file
impl = file("resources/expressions/GetOrders.ducx-xp");
```

4.3 Reference naming conventions

The next table gives an overview of suggested naming conventions for component object references. These naming conventions are only recommendations, and you are free to define your own naming conventions.

Note: A valid reference must begin with a character and must not contain any special characters except the underscore character.

Element	Recommended reference
Software component	The reference of a software component should be composed of upper case characters only. Example: APPDUCXSAMPLE
Object class	The reference of an object class should be in singular form. Use mixed case notation, starting with a capital first letter. Example: `OrderRecord`
Enumeration type, compound type	The reference of an enumeration type or a compound type should describe the purpose of a property of this type. If possible, do not include the word “Type”. Use mixed case notation, starting with a capital first letter. Example: `OrderState`
Enumeration item	The reference of an enumeration item should be composed of upper case characters only. Individual words should be separated by an underscore character. Each enumeration item should share a prefix common to all enumeration items of the enumeration type containing them. Example: `OS_SHIPPED`

Property	The reference of a property should describe the purpose of the property. You may include an abbreviation of the object class the property belongs to. Use lower case characters only for a property reference. Example: `orderdate`
Use case	The reference of a use case should be comprised of a verb describing the action performed followed by the object on which this action is carried out. Use mixed case notation, starting with a capital first letter. Example: `PrintInvoice`
ACL	The reference of an *ACL* should be in mixed case notation, starting with a capital first letter, and ending with the postfix “ACL”. Example: `OrderRecordACL`
Form	The reference of a form should be in mixed case notation, starting with the prefix “Form”. If you define different forms for administrators and end-users, you should use the postfixes “User” and “Admin”. Example: `FormOrderRecord`, `FormOrderRecordUser`, `FormOrderRecordAdmin`
Form page	The reference of a form page should be in mixed case notation, starting with the prefix “Page”. Example: `PageOrderRecord`
Other component objects	The reference of all other component objects should be in mixed case notation, starting with a capital first letter. An abbreviation of the object class may be used as a prefix. Example: `ButtonBarOrderRecord`

Table 1: Reference naming conventions

5 app.ducx Object Model Language

The purpose of the app.ducx object model language is to define the persistent object model for a software component.

Using the app.ducx object model language, you can easily define the basic elements that make up the object model:

- object classes
- properties and fields
- enumeration types
- compound types
- extensions of existing object classes and types

An object model block consists of import declarations and object model elements. The `objmodel` keyword denotes an object model block. It must be followed by the reference of your software component and curly braces.

Object model blocks can only be contained in files with a `.ducx-om` extension.

Syntax

```
objmodel softwarecomponent
{
  // Import declarations
  import softwarecomponent;
  // Object model elements (object classes, types, fields)
  ...
}
```

5.1 Defining an object class

An object class defines the abstract characteristics of an object, which include properties and the use cases that can be executed on the instances of the object class.

Syntax

```
class<metaclass> reference : baseclass {
  ...
}
```

5.1.1 Selecting the meta class

In Fabasoft Folio, each object class is an instance of a meta class that defines the characteristics of the object class itself.

The meta class is specified in angle brackets following the `class` keyword. The definition of the meta class is optional. When omitted, `COOSYSTEM@1.1:ObjectClass` is used as the meta class of the object class.

Usually, you will use one of the following meta classes for defining a new object class:

- The `COOSYSTEM@1.1:ContentObjectClass` meta class is specifically designed for object classes containing content properties or compound properties of type `COOSYSTEM@1.1:Content`. For defining an object class that is an instance of `COOSYSTEM@1.1:ContentObjectClass`, the `class` keyword must be followed by `<COOSYSTEM@1.1:ContentObjectClass>`. Full qualification of the reference is only required if `COOSYSTEM@1.1` is not imported using the `import` keyword.
- The `COOSYSTEM@1.1:ObjectClass` meta class should be used for object classes that do not contain object lists or content properties. The `class` keyword is used for defining an instance of `COOSYSTEM@1.1:ObjectClass`. `<COOSYSTEM@1.1:ObjectClass>` may be omitted as it is the default meta class.

If you want to use a different meta class, you can specify the reference of the meta class in the angle brackets following the `class` keyword.

5.1.2 Defining the base class

Each object class in Fabasoft Folio must have a base class which must either directly or indirectly be derived from object class `COOSYSTEM@1.1:Object`. This object class is the base class of all other object classes.

The base class is specified after a colon following the reference of the newly defined object class.

If you do not find a more specific base class that is appropriate for your new class, you will usually derive it from one of the following most common base classes:

- `COOSYSTEM@1.1:ContentObject` for object classes that are intended for storing content. These object classes usually have `COOSYSTEM@1.1:ContentObjectClass` as their meta class.
- `COOSYSTEM@1.1:CompoundObject` for object classes that are intended for storing object lists.

- `COOSYSTEM@1.1:ComponentObject` for object classes where the instances must be shipped with a software component.
- `COOSYSTEM@1.1:BasicObject` for other object classes when you do not find a more appropriate base class.

Cloud profile note: Base classes are restricted to `COOSYSTEM@1.1:ContentObject`, `COOSYSTEM@1.1:CompoundObject`, `COOSYSTEM@1.1:BasicObject` and classes marked as `derivable` from own and friend components.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  // New object class for storing resources that is derived from
  // COOSYSTEM@1.1:BasicObject
  class Resource : BasicObject {
    ...
  }
  // New object class for specification documents that is derived from
  // COOSYSTEM@1.1:ContentObject
  class<ContentObjectClass> SpecificationDocument : ContentObject {
    ...
  }
  // New object class for projects that is derived from
  // COOSYSTEM@1.1:CompoundObject
  class Project : CompoundObject {
    ...
  }
}
```

5.1.3 Defining miscellaneous object class aspects

Besides choosing the meta class and the base class, you can also define several other aspects of the object class, such as whether it should be abstract, programmatic, common, deriveable or compound.

5.1.3.1 Setting an object class to abstract

The `abstract` keyword allows you to define an abstract object class.

An abstract object class can only be used as a base class from which child object classes may be derived, and which cannot be instantiated itself.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class Vehicle : BasicObject {
    abstract = true;
  }
  class Car : Vehicle {
  }
}
```

5.1.3.2 Setting an object class to programmatic

Using the `programmatic` keyword, an object class can be marked as not creatable by a user via the user interface. Thus, for example, the object class is not available in the “Object” > “New” dialog but objects of the object class can be created programmatically.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class Order : BasicObject {
    programmatic = true;
  }
}
```

5.1.3.3 Marking an object class as compound

Using the `compound` keyword, an object class can be marked as compound. Only the instances of a compound object class are displayed in the tree view on the left-hand side of the Fabasoft Folio Web Client.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class Product : CompoundObject {
    compound = true;
  }
}
```

5.1.3.4 Marking an object class as a common class

Using the `common` keyword, an object class can be marked as a common class. This means that this class is creatable or usable in common locations like in a folder (i.e. in the `COOSYSTEM@1.1:objchildren` property). Using the `allowcommonclasses` keyword with a property can be marked as a property for containing common classes.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class Product : CompoundObject {
    common = true;
  }
}
```

5.1.3.5 Marking an object class as derivable

Using the `derivable` keyword, an object class can be marked as derivable. This means that classes from other software components can use this class as base class.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class Product : CompoundObject {
    derivable = true;
  }
}
```

5.2 Adding properties to an object class

Each object class can have several properties defined in which to store various data.

5.2.1 Reusing existing properties

Existing properties can be reused and referenced within an object class or within a compound type. When an existing property is referenced, this property must be available in the Fabasoft app.ducx cache.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class FileCategory : BasicObject {
    // Reusing object pointer property FSCFOLIO@1.1001:dcstate
    FSCFOLIO@1.1001:dcstate;
  }
}
```

When reusing properties, keep in mind that changes to the original property might affect your solution. For instance, the definition of the selectable object classes in the `FSCFOLIO@1.1001:dcstate` property in the example might be changed in a future version. As a result, reusing properties might result in unpredictable side-effects in some cases.

5.2.2 Defining new properties

Syntax

```
datatype reference modifiers;
```

New properties can only be defined directly within an object class or within a compound type. For each property defined using Fabasoft app.ducx, a corresponding property component object is created as part of the project's software component.

A property must be defined based on a valid Fabasoft Folio data type. Each property can either store a scalar value or a list of values of the specified data type. Add square brackets after the data type for defining a property for storing a list of items.

There are two ways of defining a new property: the simple shorthand notation and the extended notation for including *triggers* and *constraints* in a property definition.

A shorthand property definition consists of

- a data type
- square brackets as an optional list marker
- a reference
- a semicolon

Extended property definitions are covered in chapter 5.3 "Extended property definitions".

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class Project : CompoundObject {
```

```
    // Creates a new property of data type COOSYSTEM@1.1:STRINGLIST
    // with the reference APPDUCXSAMPLE@200.200:projectdescription
    string[] projectdescription;
  }
}
```

5.2.2.1 Data types

In Fabasoft Folio, every property must have a data type. Fabasoft Folio supports simple data types, enumeration types, and compound types.

5.2.2.1.1 Simple data types

Simple data types are provided by software component `COOSYSTEM@1.1`. The next table shows a list of simple data types supported by Fabasoft Folio.

Simple data type	Keyword	Maximum Size
COOSYSTEM@1.1:STRING	string	4000 characters
COOSYSTEM@1.1:BOOLEAN	boolean	
COOSYSTEM@1.1:INTEGER	integer, time, timespan	20 digits
COOSYSTEM@1.1:FLOAT	float	16 digits
COOSYSTEM@1.1:DATETIME	date, datetime	
COOSYSTEM@1.1:Currency	currency	
COOSYSTEM@1.1:CONTENT	content	
COOSYSTEM@1.1:DICTIONARY	dictionary	
COOSYSTEM@1.1:OBJECT	object	

Table 2: Simple data types

For most simple data types, a corresponding list data type is provided for storing lists of items of this type. The supported list data types are depicted in the next table.

Simple data type	Keyword
COOSYSTEM@1.1:STRINGLIST	string[]
COOSYSTEM@1.1:BOOLEANLIST	boolean[]
COOSYSTEM@1.1:INTEGERLIST	integer[], time[], timespan[]
COOSYSTEM@1.1:FLOATLIST	float[]
COOSYSTEM@1.1:DATETIMELIST	date[], datetime[]
COOSYSTEM@1.1:CONTENTLIST	content[]

COOSYSTEM@1.1:DICTIONARYLIST	dictionary[]
COOSYSTEM@1.1:OBJECTLIST	object[]

Table 3: List data types

5.2.2.1.2 Enumeration types

For information concerning the definition and extension of enumeration types, please consult chapter 5.2.2.13 “Defining enumeration types”.

5.2.2.1.3 Compound types

For information concerning the definition and extension of compound types, please consult chapter 5.2.2.15 “Defining compound types”.

5.2.2.2 Defining a string property

The `string` keyword denotes a data type that stores a string of characters and can be used for defining string properties.

The length of a string can be specified enclosed in parentheses. A string can have a maximum length of 254 characters. If the length is not specified, the default length of 254 characters is assumed.

Note: String lists can be used for multiline input fields.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class Task : BasicObject {
    // String property holding a maximum of 254 characters
    string shortdescription;
    // Sized string property holding a maximum of 2 characters
    string(2) state;
    // String list property that is displayed as a multiline input field
    string[] longdescription;
  }
}
```

5.2.2.3 Defining a Boolean property

The `boolean` keyword is used to define properties to store the Boolean values (`true` and `false`).

Note: A Boolean property can actually have three states in Fabasoft Folio as it can also be undefined.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCFOLIO@1.1001;
  class Application : Case {
    boolean approved;
  }
}
```

5.2.2.4 Defining an integer number property

The `integer` keyword denotes a simple data type and can be used to define integer number properties that store values consisting of up to 10 digits.

Integers may be signed or unsigned. Unsigned integers are capable of representing only non-negative values whereas signed integers are capable of representing negative values as well.

For defining unsigned integers use the `unsigned integer` keywords.

For an integer, its size in digits can be specified enclosed in parentheses. Valid size values are from 1 up to 10 digits. If no size value is specified, the default size of 10 digits is assumed.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class TaxReport : BasicObject {
    // Integer number property with 10 digits
    integer netchange;
    // Sized integer number property with 3 digits
    integer(3) taxrate;
    // Unsigned integer number property with 10 digits
    unsigned integer taxpayers;
    // Sized unsigned integer number property with 4 digits
    unsigned integer(4) fiscalyear;
  }
}
```

5.2.2.5 Defining a floating-point number property

The `float` keyword denotes a floating-point number data type that represents a real number and can be used to define floating-point number properties to store values consisting of up to 16 digits in total with a maximum of 9 digits of precision.

The `float` keyword can be preceded by the keyword `unsigned` to denote an unsigned float which can only represent non-negative values.

For a floating-point number property, the size in digits before and the number of digits of precision can be specified enclosed in parentheses. If no size values are specified, the default sizes of 10 digits before with 2 digits of precision are assumed.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class AssetPerformance : BasicObject {
    // Floating-point number property with 10 digits before
    // and 2 digits of precision
    float assetvalue;
    // Sized floating-point number property of a 3 digit number with
    // 2 digits of precision
    float(3,2) ytdchange;
    // Sized unsigned floating-point number property of an 8 digit number
    // with 6 digits of precision
    unsigned float(8,6) exchangerate;
  }
}
```

5.2.2.6 Defining a date property

The `date` keyword is used for defining a property to store a date value.

The `datetime` keyword is used for defining a property that stores both a date and a time value.

For the `datetime` keyword, you can append the suffix `local` to force that the value is converted to local time each time the property is accessed using a kernel interface. Alternatively, you can add the optional suffix `universal`, which is omitted by default. A `universal` date/time value is not converted to the user's local time zone when it is accessed.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class Milestone : BasicObject {
    // Date only property
    date deadline;
    // Date and time property with local time conversion enabled
    datetime documentreceivedat local;
    // Date and time property with local time conversion disabled
    datetime kickoffmeetingat;
  }
}
```

5.2.2.7 Defining a time property

The `time` keyword is used for defining a property to store a time value.

Time values are stored in seconds in an integer number property of data type `COOSYSTEM@1.1:INTEGER`. The `COOATTREDIT@1.1:CTRLTimestamp` control is used for displaying time values in time format.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  struct Session {
    string sessiontitle;
    time sessionstartat;
    time sessionendat;
  }
  class Conference : BasicObject {
    // Compound property storing a list of conference sessions
    Session[] sessions;
  }
}
```

5.2.2.8 Defining a time span property

The `timespan` keyword is used for defining a property to store a time span.

The time span is stored in an integer number property of data type `COOSYSTEM@1.1:INTEGER`. The `COOATTREDIT@1.1:CTRLTimespan` control is used for displaying the time span in days, hours, minutes, and seconds.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
```

```
  import COOSYSTEM@1.1;
  class MailPollingDefinition : BasicObject {
    // Time span for defining the e-mail polling interval
    timespan pollinginterval;
  }
}
```

5.2.2.9 Defining a currency property

The `currency` keyword is used to define a currency property that represents an amount of money in a certain currency.

For a currency property, the size in digits before and the number of digits of precision can be specified enclosed in parentheses.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class Project : CompoundObject {
    // Currency property for storing an amount of money in a currency
    // that can be selected by the user
    currency projectvalue;

    // Sized currency property of a 12 digit number with
    // 4 digits of precision
    currency(12,4) foreignchange;
  }
}
```

5.2.2.10 Defining a content property

A content property that is defined using the `content` keyword can store any kind of binary data, such as a Microsoft Word or LibreOffice document.

A file can be imported into a content property from the file system, and conversely content that is stored in a content property can be exported to the file system.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class Project : CompoundObject {
    // Content property for storing a project plan in a binary format
    content projectplan;
  }
}
```

5.2.2.11 Defining a dictionary property

A dictionary is a list of key-value pairs. Each key-value pair consists of a string which is used as a key for addressing the dictionary entry, and a value or a list of values of the same Fabasoft Folio data type.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class StatusLog : BasicObject {
```

```
    // Dictionary property for storing key-value pairs
    dictionary valuedict;
  }
}
```

5.2.2.12 Defining an object pointer property

An object pointer property can store a pointer to an object or a list of pointers to objects.

No explicit keyword is required for defining an object pointer property. Instead, the object class of the objects that shall be referenced by the new object pointer property is used as data type.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class PeerGroup : BasicObject {
    // Object pointer for referencing a user object
    User responsibleuser;
    // Object list for storing a list of user objects
    User[] groupmembers;
  }
}
```

By default, instances of all object classes that are directly or indirectly derived from the object class provided in the definition can be selected in the object pointer. A default object pointer property will also allow the creation of new object instances within the property. However, for object pointer properties, you can provide an extended definition of the object classes allowed and not allowed in the property in order to limit the selectable and creatable items.

The `allow` keyword can be used to define a list of object classes that should be allowed in the object pointer property, and the `exclude` keyword can be used for excluding object classes. Both the object classes listed in the `allow` block and in the `exclude` block must be either directly or indirectly derived from the object class provided in the object pointer property definition.

For each of the object classes listed within an `allow` block, the `create` keyword can be added after the reference of the object class to allow the creation of instances of this object class in the object pointer property. If the `create` keyword is omitted, no instances can be created directly within the object pointer property.

Object classes listed within an `allow` or `exclude` block must be separated by semicolons.

The `allowcommonclasses` keyword defines that the object pointer property can contain all common object classes (all object classes with `common` set to `true`).

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCFOLIO@1.1001;
  class Department : BasicObject {
    // This definition creates an object pointer for referencing a
    // person object; it will not allow the creation of a new person
    // object within the object pointer
    Person deptmanager {
      allow {
        Person;
      }
    }
    // This definition creates an object list for storing objects that
    // can contain persons and employees but no contact persons; only
```

```
    // employee objects can be created in this list
    Person[] deptmembers {
      allow {
        Person;
        Employee create;
      }
      exclude {
        ContactPerson;
      }
    }
    // This definition creates an object list for storing all
    // common documents classes
    ContentObject[] documents {
      allowcommonclasses = true;
    }
  }
}
```

5.2.2.13 Defining enumeration types

In contrast to properties, which are defined directly within an object class, an enumeration type can only be defined as a separate `objmodel` model element.

A custom enumeration type can be defined using the `enum` keyword which must be followed by a reference and a block containing the permissible enumeration items separated by commas.

Each enumeration item can be assigned a specific integer value. For enumeration items that are not assigned to a specific integer value, the compiler automatically generates integer values. To do so, the compiler simply increments the integer value of the preceding enumeration item by 1, starting with value 1 for the first item if it does not have an assigned value.

It is suggested that you always assign an integer value at least for the first enumeration item of an enumeration type, since using different number ranges for different enumeration types increases database performance. For example, you might want to assign a starting value of 1,000 for the first enumeration type in your software component, and a starting value of 10,000 for the second enumeration type, and so on.

Note: It is highly recommended to explicit assign integer values to avoid error on later extensions of the enumeration type.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  // Enumeration type consisting of three enumeration items which are
  // assigned system generated integer values, starting with 1
  // This kind of writing is not recommanded, because of the automatic numeration
  // it is highly error-prone on extension or expansions
  enum ProjectState {
    PS_PLANNED,
    PS_UNDERWAY,
    PS_COMPLETED
  }
  // Enumeration type consisting of four enumeration items with
  // explicitly assigned integer values
  enum ShipType {
    ST_DESTROYER = 100,
    ST_CRUISER = 200,
    ST_BATTLESHIP = 300,
    ST_SUBMARINE = 400
  }
  // Enumeration type consisting of six enumeration items with an
  // explicitly assigned starting value; the remaining enumeration items
  // get assigned integer values 5001, 5002, 5003 and so on.
  enum NewEnglandStates {
    ST_CONNECTICUT = 5000,
    ST_NEWHAMPSHIRE,
```

```
    ST_MAINE,
    ST_MASSACHUSETTS,
    ST_RHODEISLAND,
    ST_VERMONT
  }
  // Enumeration type consisting of enumeration items with partially
  // assigned integer values
  enum DrinkType {
    DT_SHOT = 800,        // integer value 800 manually assigned
    DT_LONGDRINK = 850,   // integer value 850 manually assigned
    DT_SOUR,              // integer value 851 generated by compiler
    DT_FIZZ,              // integer value 852 generated by compiler
    DT_HIGHBALL,          // integer value 853 generated by compiler
    DT_SOFTDRINK = 900,   // integer value 900 manually assigned
    DT_OTHER              // integer value 901 generated by compiler
  }
}
```

Note: Existing enumeration types can be extended with additional enumeration items. For details on how to extend an existing enumeration type with additional enumeration items, please consult chapter 5.5.2 “Extending an existing enumeration type”.

5.2.2.14 Defining an enumeration property

As with an object pointer property, no explicit keyword is required for defining an enumeration property. Instead, you must provide the enumeration type which you want to assign to the new enumeration property.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  enum Transmission {
    TR_AUTOMATIC = 10000,
    TR_MANUAL = 10001
  }
  enum Options {
    OP_LEATHERSEATING = 20000,
    OP_AIRCONDITIONING = 20001,
    OP_FOGLAMPS = 20002
  }
  class Car : BasicObject {
    // Enumeration property that can store a single enumeration item
    Transmission transmission;
    // Enumeration property that can store a list of enumeration items
    Options[] options;
  }
}
```

5.2.2.15 Defining compound types

A compound type is a structure that can contain a list of properties of any valid Fabasoft Folio data type. Similarly to enumeration types, compound types can be defined or extended within an `objmodel` block.

5.2.2.15.1 Defining a new compound type

A custom compound type can be defined using the `struct` keyword that must be followed by a reference and a block containing the properties that should become part of the compound type separated by semicolons.

You can either reference existing properties or define new properties directly within a compound type.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;

  // Compound type for storing publications
  struct Publication {
    string title;                   // new property for storing the title
    userfirstname;                  // reuse first name property
    usersurname;                    // reuse surname property
    integer(4) yearofpublication;   // new year of publication property
    string isbn;                    // new isbn property

    typecast = TypeCastPublication; // type cast
  }
}
```

5.2.2.15.2 Defining the key for a compound type

For a compound type, you can define one or more key properties. The key is evaluated for determining unique entries in a property using the compound type.

The properties making up the key must be referenced in a block that is assigned to `COOSYSTEM@1.1:typeuniqueattrs`.

In addition to setting the key properties for a compound type, you can also define that compound properties using this compound type can contain unique entries only. To enforce unique keys, `COOSYSTEM@1.1:typelistunique` must be set to `true`.

In the following example, the `APPDUCXSAMPLE@200.200:isbn` property is specified as the key for compound type “Publication”. Additionally, the `COOSYSTEM@1.1:typelistunique` is set to `true` so the compound type will only allow entries with a unique ISBN.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;

  struct Publication {
    string isbn;
    string title;
    typelistunique = true;     // require unique entries
    typeuniqueattrs = {        //
      isbn                     // list of key properties
    }                          //
  }
}
```

Note: You can also extend existing compound types with new properties. For details on how to extend an existing compound type with new properties, please consult chapter 5.5.3 “Extending an existing compound type”.

5.2.2.15.3 Defining a type cast for a compound type

Optionally a type cast may be defined. In expressions the type cast allows you to assign values directly to properties of the compound type, based on the type of the value without need to explicitly specify the property. The type cast defines which type should be assigned to which property. The `TypeCastPrototype` has two parameters: `source` and `target`.

In the following example string lists are assigned to `features` and integers and floats are assigned to `productioncosts`. In an expression you can make following assignment: `Feature feature = 300;`. The result is the same as when writing `feature.productioncosts = 300;`.

Note: The direct assignment to the compound property `productioncosts` works because for currencies a default type cast is provided by default.

Example

app.ducx Object Model Language

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  struct Feature {
    currency productioncosts;
    string[] features;
    typecast = TypeCastFeature;
  }
}
```

app.ducx Use Case Language

```
usecases APPDUCXSAMPLE@200.200
{
  TypeCastFeature(parameters as TypeCastPrototype) {
    variant Object {
      impl = expression {
        Feature target = {};
        if (typeof(source) == #STRINGLIST) {
          target.features = source;
        }
        else if (typeof(source) == #INTEGER || typeof(source) == #FLOAT) {
          target.productioncosts = source;
        }
        target;
      }
    }
  }
}
```

5.2.2.15.4 Using a primary property for direct assignments

For some compound properties it is likely that only one property of the compound property is set (e.g. `FSCVAPP@1.1001:applytofield`). For this case a property of the compound property can be marked by setting `COOSYSTEM@1.1:typeprimaryattr`. Additionally the predefined type cast `COOSYSTEM@1.1:TypeCast` has to be used.

In the following example title is marked as primary property thus assignments like `Publication publication = "An Introduction to Fabasoft app.ducx";` are possible.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  struct Publication {
    string title;
    userfirstname;
    usersurname
    integer(4) yearofpublication;
    string isbn;
    typecast = TypeCast;
    typeprimaryattr = title;
  }
```

```
}
```

For following compound types primary properties are defined:

Compound type	Primary property
FSCVAPP@1.1001:Branch	FSCVAPP@1.1001:identopt
FSCVAPP@1.1001:ApplyToField	FSCVAPP@1.1001:identopt
COOSYSTEM@1.1:LanguageStringList	COOSYSTEM@1.1:langstring
COOSYSTEM@1.1:LanguageContentList	COOSYSTEM@1.1:langcontent
COOSYSTEM@1.1:Content	COOSYSTEM@1.1:contcontent
COOSYSTEM@1.1:Currency	COOSYSTEM@1.1:currvalue

Table 4: Compound types and corresponding primary properties

Note: Direct assignments can not only be utilized in expressions but also in domain specific languages for initializing values or defining instances.

5.2.2.16 Defining a compound property

In the same way as for an enumeration property, the definition of a compound property does not require a special keyword. Instead, the compound type is used as data type for the new compound property.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  struct OrderPosition {
    integer quantity;
    Product product;
  }
  class Product : BasicObject {
    ...
  }
  class Order : CompoundObject {
    // Compound property for storing a list of order positions
    OrderPosition[] orderpositions;
    // Compound property that is reusing the compound type
    // COOSYSTEM@1.1:Content
    Content orderdescription;
  }
}
```

5.2.2.17 Defining a link property

A link property allows you to create a bidirectional relationship between objects. There are two ways to establish such a relationship. On the one hand the `backlink` keyword can be used to create explicitly back link objects that manage the bidirectional relationship, on the other hand just the `link` keyword can be used. In the second case no back link objects are created.

The `backlink` keyword is used for defining a back link property. It must be followed by a reference and curly braces.

Use `link` when referencing a link or back link property from an object pointer property in order to establish a bidirectional relationship.

Back link properties must be directly assigned to an object class. You cannot use the back link properties in a compound type. However, an object pointer property referencing a back link property can also be part of a compound type.

In the following example, an order has an object pointer property pointing to an invoice. However, an invoice also needs to know to which order it belongs to. Therefore, a back link property is defined for pointing back to the order. The back link ensures that the integrity of the relationship between linked objects is maintained automatically. Whenever the order's invoice object pointer property is changed, the change is reflected in the invoice.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class Order : CompoundObject {
    Invoice orderinvoice {
      link = invoiceorder;
    }
  }
  class<ContentObjectClass> Invoice : ContentObject {
    backlink<Order> invoiceorder readonly;
  }
}
```

The following example shows the use of only the `link` keyword. The links ensure that the integrity of the relationship between linked objects is maintained automatically. Whenever the order's invoice object pointer property is changed, the change is reflected in the invoice and vice versa.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class Order : CompoundObject {
    Invoice orderinvoice {
      link = invoiceorder;
    }
  }
  class<ContentObjectClass> Invoice : ContentObject {
    Order invoiceorder {
      link = orderinvoice;
    }
  }
}
```

5.2.3 Property modifier prefixes

When defining new properties within an object class or within a compound type, you can add so-called property modifier prefixes before the property's reference.

`unique` is currently the only available modifier prefix, which can be used to define that values of a list property have to be unique.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
```

```
  import FSCFOLIO@1.1001;
  class Order : CompoundObject {
    unique OrderPosition[] orderpositions;
  }
}
```

5.2.4 Property modifier suffixes

When defining new properties within an object class or within a compound type, you can add so-called property modifier suffixes (also referred to as modifiers) after the property's reference. Modifiers allow you to change the appearance of a property as well as some other aspects pertaining to the representation of property values.

Table 5 contains a list of supported modifiers along with a brief description.

Modifier	Description
`not null`	The property must contain a value.
`readonly`	The property is not changeable in any circumstance.
`readonly(ui)`	The property is not changeable in the user interface.
`readonly(inst)`	The property is not changeable when the software component of the component object is in the "Installed" state.
`volatile`	The property is not cached by the Fabasoft Folio Kernel.
`invisible`	The property is not visible in the user interface.

Table 5: Property modifier suffixes

If multiple modifiers are appended to a property, they must be separated by whitespaces.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCFOLIO@1.1001;
  enum OrderState {
    OS_PENDING = 100,
    OS_APPROVED = 101,
    OS_DISCARDED = 102,
    OS_SHIPPED = 103,
    OS_COMPLETED = 104,
    OS_ARCHIVED = 105
  }
  class Order : CompoundObject {
    datetime orderdate readonly(ui);
    OrderState orderstate readonly(ui);
    Person ordercustomer not null {
      allow {
        Person;
      }
    }
    OrderPosition[] orderpositions not null;
    integer orderpositioncount volatile readonly;
    Invoice orderinvoice readonly(ui);
    currency ordertotal readonly(ui);
  }
}
```

5.3 Extended property definitions

An extended property definition allows you to define initialization values, to assign triggers to and to define constraints for your properties.

The following syntax example depicts the structure of extended property definitions.

Syntax

```
datatype reference modifiers {
  // Definition of an initialization value
  init = initvalue;

  // Definition of the access type required to read the property
  accget = getaccesstype;
  // Definition of the access type required to write the property
  accset = setaccesstype;

  // Trigger that is called when the property is created
  ctor = constructortriggeraction;
  // Trigger that is called when the value of the property is read
  get = getvaluetriggeraction;
  // Trigger that is called when the value of the property is written
  set = setvaluetriggeraction;
  // Trigger that is called when the value of the property is copied
  copy = copytriggeraction;

  // Value constraint
  value = expression {
    ...
  }
  // Filter constraint
  filter = expression {
    ...
  }
  // Validation constraint
  validate = expression {
    ...
  }
  // User interface change constraint
  uichange = expression {
    ...
  }
}
```

5.3.1 Initializing a property with a value

Using the `init` keyword, you can assign an initialization value to a property. The property is initialized with this value when you create a new instance of the object class the property is assigned to.

In addition to static initialization values, you can also use app.ducx expression language to calculate the value used for initializing a property. An `expression` block is required to define an initialization expression.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class Order : CompoundObject {
    datetime orderdate readonly(ui) {
      // initialization with an expression block
      // example of a static initialization for datetime:
      // init = 2010-10-10T09:13:27;
      init = expression {
```

```
        coonow;
      }
    }
    OrderState orderstate readonly(ui) {
      init = OS_PENDING;
    }
    string ordershortdescription {
      init = "Please enter a description text!";
    }
    integer(3) orderitems readonly(ui) {
      init = 0;
    }
  }
}
```

5.3.2 Protecting a property with access types

A property can be protected with an access type for reading the property value and with an access type for changing the property:

- The `accget` keyword is used to assign an access type for reading the property value. When protected by an access type for reading the property value, the property is only displayed if the access type is granted to the user by the object's ACL.
- The `accset` keyword is used to assign an access type for changing the property value. When protected by an access type for changing the property value, the property may only be changed if the access type is granted to the user by the object's ACL.

Both the access type for reading and for changing the property value are optional.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class Order : CompoundObject {
    Product[] orderedproducts not null {
      accget = AccTypeReadComp;
      accset = AccTypeChangeComp;
    }
  }
}
```

5.3.3 Assigning triggers to a property

Triggers are actions that are invoked or "fired" automatically when a predefined event occurs. Each trigger has assigned a specific Fabasoft Folio action prototype listing the parameters that are passed to the implementation of the trigger action when it is called.

Keyword	Description
`ctor`	The constructor trigger is fired when the property is created. A constructor trigger action must have assigned the prototype `COOSYSTEM@1.1:AttrConstructorPrototype`.
`linector`	The line constructor trigger is fired when a new entry is created in a compound property. A line constructor trigger action must have assigned the prototype `COOSYSTEM@1.1:AttrLineConstructorPrototype`.
`dtor`	The destructor trigger is fired when the property is destroyed. A destructor trigger action must have assigned the prototype

	COOSYSTEM@1.1:AttrDestructorPrototype.
get	The get value trigger is fired after the property value is read from the database. A get value trigger action must have assigned the prototype COOSYSTEM@1.1:AttrGetPrototype.
set	The set value trigger is fired before the property value is written to the database. A set value trigger action must have assigned the prototype COOSYSTEM@1.1:AttrSetPrototype.
copy	The copy trigger is fired when an object containing the property is duplicated. A copy trigger action must have assigned the prototype COOSYSTEM@1.1:AttrCopyPrototype.
display	The display trigger is fired to format the property value for being displayed in a list view column. A display trigger action must have assigned the prototype COOSYSTEM@1.1:AttrGetDispPrototype.
search	The search trigger is fired when searching for values in the property. A search trigger action must have assigned the prototype COOSYSTEM@1.1:AttrSearchPrototype.
getver	The get versioned value trigger is fired when a version of the property value is read. A get versioned value trigger action must have assigned the prototype COOSYSTEM@1.1:AttrGetVersionPrototype.
fixver	The fix version trigger is fired when a version of the property value is created. A fix version trigger action must have assigned the prototype COOSYSTEM@1.1:AttrFixVersionPrototype.
delver	The delete version trigger is fired when a version of the property is deleted. A delete version trigger action must have assigned the prototype COOSYSTEM@1.1:AttrDelVersionPrototype.
restver	The restore version trigger is fired when a version of the property value is restored. A restore version trigger action must have assigned the prototype COOSYSTEM@1.1:AttrRestVersionPrototype.
archive	The archive trigger is fired when the property value is archived. An archive trigger action must have assigned the prototype COOSYSTEM@1.1:AttrArchivePrototype.
restore	The restore trigger is fired when the property value is being restored from an archive. A restore trigger action must have assigned the prototype COOSYSTEM@1.1:AttrRestArchivePrototype.

Table 6: Property triggers supported by Fabasoft app.ducx

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class Order : CompoundObject {
    // The value of the ID property of an order must not be copied when
    // an order object is duplicated
    unsigned integer(6) orderid {
      copy = NoOperation;
    }
```

```
  }
}
```

For further information on how to implement trigger actions, please consult chapter 8.8 “Implementing triggers”.

5.3.4 Assigning constraints to a property

Fabasoft Folio uses integrity and value constraints for calculating and validating values and to prevent invalid data entry into a property.

Fabasoft app.ducx currently supports following types of constraints:

- Value constraints are useful for automatically calculated property values such as the grand total of an order or the number of order positions.
- Filter constraints allow you to limit the selectable values for object pointer properties.
- Validation constraints are typically expressed as conditions or measurements that must remain true as the user uses your solution.
- User interface change constraints are evaluated when the value of an object pointer property is changed by the user, and can be used to trigger the re-evaluation of filter constraints or to set or clear the values of other properties.

5.3.4.1 Defining value constraints for a property

A value constraint can be defined for the automatic calculation of a property value.

The keyword `value` is used for defining a value constraint where the app.ducx expression language must be used for the implementation.

Value constraints only make sense for calculated properties that cannot be changed by the user. Therefore, in most cases, properties with value constraints should also be set to `volatile` and `readonly` using appropriate property modifiers. If the property is not read-only the app.ducx expression must have a result that can store the changeable value (e.g. a property). Additionally value constraints cannot be used within compound properties.

Please note that volatile properties do not show up in the search dialog box by default unless you attach a search trigger action to the property as demonstrated in the example.

When implementing a value constraint in app.ducx expression language, the current object can be accessed using the local scope `this` or global scope `::this`. The calculated value that should be displayed in the property must be passed back as the return value of the expression.

In this example, the total value of all individual order positions of an order is calculated and displayed in a property with a value constraint.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  struct OrderPosition {
    Product product not null;
    unsigned integer quantity not null;
  }
  class Product : BasicObject {
    mlname;
    string[] productdescription;
    currency unitprice;
  }
  class Order : CompoundObject {
```

```
    OrderPosition[] orderpositions;
    currency ordertotal readonly volatile {
      value = expression {
        // Add up the price of the order positions
        for (OrderPosition @position : orderpositions) {
          Product @product = @position.product;
          if (@product != null) {
            currency @total += @product.unitprice * @position.quantity;
          }
        }
        // Return the grand total
        return @total;
      }
    }
  }
}
```

5.3.4.2 Defining filter constraints for a property

Filter constraints allow you to limit the values that are selectable in an object pointer property.

The keyword `filter` or `searchfilter` is used for defining a filter constraint. The app.ducx expression language must be used for implementing the filter constraint. Search filters are used to reduce available objects of object pointer properties in context of a search.

There are two distinct variants of filter constraints that have different scopes depending on the data type of the return value of the filter expression:

- Object list filter constraints
- Boolean filter constraints

Note:

- `COOSYSTEM@1.1:attrfilterexpr`
  A Fabasoft app.ducx Expression that determines or filters the possible selection of objects in an object pointer property. If used in conjunction with `COOSYSTEM@1.1:AttrFilterCheckSet` this Fabasoft app.ducx Expression is used to test the value of a property (of any type).
- `COOSYSTEM@1.1:attrsearchfilterexpr`
  This filter is evaluated if `CAM_SEARCH` is passed in the parameter `mode` to the action `COOSYSTEM@1.1:LocalObjectsGet`. Additionally the transaction variable `TV_SEARCHOBJCLASS` of type `OBJECT` is provided to be able to specify already chosen object classes in the search filter. The transaction variable contains the chosen object class (several in case of a quick search).

5.3.4.2.1 Object list filter constraints

An object list filter constraint is an expression returning a list of objects. For properties with an object list filter constraint, users can only select values out of the list of permitted values.

For an object list filter constraint, the object containing the object pointer property can be accessed using the local scope `this`. The global scope `::this` contains a list of objects that should be filtered. If empty, the Fabasoft app.ducx Expression should retrieve a meaningful default for objects that are usable as a value for the property.

5.3.4.2.2 Boolean filter constraints

A Boolean filter constraint is evaluated on each of the objects that would be selectable in an object pointer property based on the definition of allowed object classes for the property. If the filter expression returns `true`, the object is included in the list of selectable values displayed to users. Otherwise, the object is filtered out and cannot be selected.

For a Boolean filter constraint, the local scope `this` contains the object that is to be selected and the global scope `::this` contains the object containing the object pointer property.

For instance, if a Boolean filter constraint is defined for an object pointer property for selecting users in a Fabasoft Folio Domain containing 300 user objects, the filter expression would be evaluated up to 300 times as it is evaluated for each user object.

In the example, there are three object classes: product, vendor, and order. An instance of vendor has a list of products sold by the vendor. An order contains an object pointer property pointing to a vendor, and a list of products ordered from this vendor. For the order's vendor object pointer property, a Boolean filter constraint has been defined only allowing the user to select vendors that actually sell products. Moreover, the list of ordered products has attached an object list filter constraint that limits the selectable products to the list of products offered by the vendor referenced in the vendor object pointer property.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class Product : BasicObject {
    string productid not null;
  }
  class Vendor : CompoundObject {
    Product[] offeredproducts;
  }
  class Order : CompoundObject {
    Vendor vendor not null {
      filter = expression {
        offeredproducts != null;
      }
    }
    Product[] orderedproducts not null {
      filter = expression {
        vendor != null ? vendor.offeredproducts : null;
      }
    }
  }
}
```

When a value, filter, validation or user interface change constraint is evaluated, the transaction variables of software component `COOSYSTEM@1.1` listed in the next table provide you with information on the path to the currently selected property.

Transaction Variable	Description
`TV_ATTRPATHATTRDEFS`	`TV_ATTRPATHATTRDEFS` contains the property path to the currently selected property. For example, if the object pointer property `APPDUCXSAMPLE@200.200:product` in the third row of the compound property `APPDUCXSAMPLE@200.200:orderpositions` of an order is selected, `TV_ATTRPATHATTRDEFS` contains an object list consisting of the elements `APPDUCXSAMPLE@200.200:orderpositions` and `APPDUCXSAMPLE@200.200:product`.
`TV_ATTRPATHINDICES`	`TV_ATTRPATHINDICES` contains the zero-based indices of the selected rows to the currently selected property. For example, if the object pointer property `APPDUCXSAMPLE@200.200:product` in the third row of the compound

	property `APPDUCXSAMPLE@200.200:orderpositions` of an order is selected, `TV_ATTRPATHINDICES` contains an integer list consisting of the elements `2` and `0`.

Table 7: Transaction variables exposing path information

The following example demonstrates how to access the transaction variables provided by software component `COOSYSTEM@1.1`.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCFOLIO@1.1001;
  class Product : BasicObject {
    mlname;
    string[] productdescription;
    currency unitprice;
  }
  class Vendor : CompoundObject {
    Product[] offeredproducts;
  }
  struct OrderPosition {
    Vendor vendor;
    Product product {
      // This filter constraint limits the selectable products so that only
      // products offered by the selected vendor can be selected in the
      // APPDUCXSAMPLE@.200.200:product property
      filter = expression {
        Product[] @products = null;
        integer[] @line = #TV.TV_ATTRPATHINDICES[0];
        Object @order = this;
        OrderPosition @orderpositionaggr = @order.orderpositions[@line];
        if (@orderpositionaggr != null) {
          Object @vendor = @orderpositionaggr.vendor;
          if (@vendor != null) {
            @products = @vendor.offeredproducts;
          }
        }
        @products;
      }
    }
    integer quantity;
  }
  class Order : BasicObject {
    OrderPosition[] orderpositions;
  }
}
```

5.3.4.3 Defining validation constraints for a property

A validation constraint is an app.ducx expression that is evaluated to check whether the value entered into the property is valid in order to prevent invalid data entry into a property.

The expression must return a Boolean value. The return value `true` signals that the value entered by the user is valid. If the expression returns `false`, an error message is displayed on the form page containing the property requiring the user to correct the entered value.

Alternatively, the expression can also throw an exception for displaying a custom error message in case the validation fails.

The local scope contains the vApp state dictionary. The global scope contains a dictionary holding the keys listed in the next table.

Key	Description
`value`	The `value` key stores the current value of the property.
`root`	The `root` key stores the object containing the property.
`parent`	This key stores the parent of the property. If the property is part of a compound type, the parent is holding the compound property. If the property is not embedded within a compound property, the `parent` key holds the object containing the property. Thus, for properties that are not part of a compound property, the value stored in the `parent` key is equal to the value stored in the `root` key.
`attribute`	The `attribute` key contains the property definition.

Table 8: Global scope for validation expressions

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCFOLIO@1.1001;
  class Project : CompoundObject {
    // Validation constraint ensures that project priority must be in
    // the range from 1 to 10
    unsigned integer(2) priority not null {
      validate = expression {
        ::value >= 1 && ::value <= 10;
      }
    }
    // COOSYSTEM@1.1:CheckWorkDay throws an exception to show a custom
    // error message if the date is not set to a work day in the
    // future
    datetime kickoffmeeting not null {
      validate = expression {
        ::root.CheckWorkDay(datetime(::value).local, true);
        return true;
      }
    }
    // Validation constraint throwing a custom error message
    date deadline not null {
      validate = expression {
        if (::value > coonow) {
          throw #COODESK@1.1:InValidDate;
        }
        else {
          true;
        }
      }
    }
  }
}
```

Defining user interface change constraints for a property

A user interface change constraint can be defined in order to trigger the re-evaluation of filter constraints or to set or clear the values of other properties.

The keywords `uichange` (value gets changed), `uisearchchange` (value gets changed in context of a search) and `uiselchange` (object gets selected) are used for defining a user interface change constraint. The `weight` constraint can be used to change the background color of an object. The app.ducx expression language must be used to implement a user interface change constraint.

The local scope contains the vApp state dictionary. When implementing a user interface change constraint in app.ducx expression language, a dictionary holding the keys listed in the previous table is made available in the global scope `::this`. The expression must return the value `true` to trigger a round-trip to the Fabasoft Folio Web Service, which is necessary for the re-evaluation of filter constraints.

In the following example, the invoice date, the payment date and the processing state properties are cleared if the object pointer property referencing the order is changed.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class<ContentObjectClass> Invoice : ContentObject {
    Order invoiceorder {
      uichange = expression {
        ::root.invoicestate = "IS_PENDING";
        ::root.invoicedate = null;
        ::root.invoicepaymentdate = null;
      }
    }
    date invoicedate;
    date invoicepaymentdate;
    InvoiceState invoicestate readonly(ui);
  }
}
```

5.3.5 Object pointer property containing children

Object pointer properties can be set as `child`, if the referenced object values do not exist as separate entities; but have a meaning only as part of the container object. For child properties the following are the default values:

- `copy` is `COOSYSTEM@1.1:NoOperation`,
- `accget` is `COOSYSTEM@1.1:AccTypeReadComp`,
- `accset` is `COOSYSTEM@1.1:AccTypeChangeComp`,
- `fixver` is `COOSYSTEM@1.1:AttrChildrenFixManualVersion`,
- `COOSYSTEM@1.1:attrrecursion` is `false`.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class Order : BasicObject {
    OrderPosition[] orderpositions {
      child = true;
      ...
    }
  }
}
```

5.3.6 Object pointer property describing a hierarchy

Object pointer properties can be set as `hierarchy`, if the referenced object values are of a similar class as the container so that the values are in a hierarchical context. When selecting a value for a hierarchy property in a search form, the hierarchy relations can be specified in the user interface.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class Order : BasicObject {
    Order baseorder {
      hierarchy = true;
      ...
    }
    Order[] suborders {
      hierarchy = true;
      ...
    }
  }
}
```

5.4 Defining fields

A field is a property that does not belong to an object class. Thus, fields may be used to temporarily compute values that are not persistently stored.

The `fields` keyword is used to define fields. It must be nested within the `objmodel` block.

Each field consists of a datatype followed by a reference and a semicolon.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  fields {
    User orderagent;
    currency ordertotal;
    string orderstatusmsg;
  }
}
```

5.5 Extending existing component objects

Fabasoft app.ducx allows you to extend existing object classes, enumeration types, and compound types that belong to another software component.

Cloud profile note: The extension of object classes, enumeration types, and compound types that belong to another non-friend software component is not allowed.

5.5.1 Extending an existing object class

You can extend an existing object class belonging to another software component with new properties of your software component.

Please note that you cannot extend object classes not belonging to your software component with properties not belonging to your software component. However, you can reuse existing properties of your software component for extending an existing object class, even if the object class belongs to a different software component.

When extending an existing object class, be sure to add a reference to the software component of this object class. Otherwise, your project will not compile.

The `extend` keyword must be used for denoting an extension, followed by the `class` keyword and the reference of the object class to be extended. Then the new properties that should be added to the object class can be defined or referenced inside the `class` block.

Syntax

```
extend class reference {
  ...
}
```

In the following example, object class `FSCFOLIO@1.1001:Organisation` is extended with a list of projects.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCFOLIO@1.1001;
  class Project : CompoundObject {
    ...
  }
  // FSCFOLIO@1.1001:Organisation is extended with the object pointer
  // list APPDUCXSAMPLE@200.200:orgprojects
  extend class Organisation {
    Project[] orgprojects;
  }
}
```

5.5.2 Extending an existing enumeration type

With the `extend` keyword, followed by the `enum` keyword and a block holding the enumeration items, you can add enumeration items to an existing enumeration type belonging to another software component.

Syntax

```
extend enum reference {
  ...
}
```

When extending an existing enumeration type, be sure to assign integer values for the new enumeration items that will not conflict with possible entries that might be added in a future version.

For example, the enumeration type `FSCFOLIO@1.1001:DocState` already contains enumeration items with the integer values of 10, 20, 30 and 40. A logical continuation of this sequence would include the integer values 50, 60, 70 and so. Therefore, you should avoid these integer values when extending the enumeration type.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  // Extend enumeration type FSCFOLIO@1.1001:DocState with a new
  // enumeration item for approved documents
  extend enum FSCFOLIO@1.1001:DocState {
    DS_APPROVED = 30000
  }
}
```

5.5.3 Extending an existing compound type

The `extend` keyword followed by the keyword `struct` and a block holding the properties allows you to extend compound types of other software components with properties belonging to your software component.

Syntax

```
extend struct reference {
  ...
}
```

When extending a compound type of another software component, reusing properties that belong to other software components is not permitted in the compound type.

Furthermore, be aware that compound types belonging to software component `COOSYSTEM@1.1` cannot be extended.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  // Extend compound type FSCFOLIO@1.1001:Address with a new string
  // property for storing the state
  extend struct FSCFOLIO@1.1001:Address {
    string(2) addrstate;
  }
}
```

5.6 Defining relations

To define a connection between two classes use the keyword `relation`. This relation can have 1:1, 1:n or n:m cardinality. A relation can have its own properties.

A special case is a “self-relation”, when the two related classes are identic.

Syntax

```
// relation without properties
// 1:1 relation
relation<classreferencea, classreferenceb> relationreference;
// relation with properties
// 1:n relation, use [] to denote cardinality
relation<classreferencea, classreferenceb[]> relationreference {
  ...
}
//self-relation
relation<classreference> relationreference;
```

In order to facilitate the use of relations app.ducx implicitly defines the necessary attributes to have access from the relation instance to the class instances and from class instances to the relation instance. For the `relation<ClassA, ClassB> Relation` the implicitly defined attributes have the following names:

Name	Container	Type
relation_classa	Relation	ClassA
relation_classb	Relation	ClassB

classa_relation	ClassA	Relation
classb_relation	ClassB	Relation

Table 9: Implicitly defined attributes for a relation

In case of the self-relation `relation<Class> Relation`:

Name	Container	Type
relation_class1	Relation	Class
relation_class2	Relation	Class
class_relation	Class	Relation

Table 10: Implicitly defined attributes for a self-relation

If `ClassA` is marked as multiple, then `classb_relation` is a list: instances of `ClassB` can have a set of `Relation` instances with different instances of `ClassA`.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  //1:n relation:
  //more persons are hired at a company, one person can be hired at one company
  relation<Company, Person[]> Employee {
    date membersince;
    integer contractid;
  }
  //n:m relation:
  //more persons can work on a project, one person can work on multiple projects
  relation<Project[], Person[]> ProjectMember;
  // self-relation
  // 1:1 relation: each person can be married to exactly one other person
  relation<Person> Marriage;
```

For instance, assume at a company several employees can be hired. This can be modeled as a 1:n relation called `Employee` between the `Company` and `Person` class. In our case, the `contractid` and `membersince` properties could describe the `Employee` relation.

5.7 Defining and extending component object instances

5.7.1 Defining a new component object instance

Using the `instance` keyword, you can define instances of component objects that become part of your software component. You can only define instances of object classes that are either directly or indirectly derived from `COOSYSTEM@1.1:ComponentObject`.

The `instance` keyword must be followed by the object class of which an instance should be created, and by the reference of the new instance. You can use generic assignment statements inside the `instance` block to assign values to the properties of the instance.

Syntax

```
instance objectclass reference {
  ...
```

```
}
```

For maintaining a well-organized project structure, it is recommended that you create a separate `.ducx-om` file for defining instances of component objects although – from a syntactic point of view – you could define all object model-related elements within a single `.ducx-om` file.

Cloud profile note: Only component objects can be instantiated.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCFOLIO@1.1001;
  import COOTC@1.1001;
  import COOAR@1.1;
  import FSCCHE@1.1;
  // Definition and initialization of a document category
  instance ComponentDocumentCategory DocumentCategoryOrder {
    dcshortform = "Order";
    categorycommon = false;
  }
  // Definition and initialization of an active report
  instance ActiveReport_Web CaseReport {
    content = file("resources/test.txt");
    transform<transapp, transpassive, transencoding, transactive,
      transembplain<textembbeg, textembend>> = {
      {APP NONE, FORMAT TXT, windows1252, "JavaScript", {{"<%", "%>"}}}
    }
  }
}
```

5.7.2 Extending an existing component object instance

Syntax

```
extend instance reference {
  ...
}
```

Using the `extend instance` keywords, it is possible to extend an existing component object.

The `extend instance` keywords must be followed by the reference of the component object to be extended. You can use generic assignment statements inside the `extend instance` block to assign values to the properties of the instance.

Cloud profile note: For instance extensions several restrictions apply. For a detailed overview, please refer to chapter 14.7.1 “Object Model Language”.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  // Exclude order and invoice objects from the list of objects allowed
  // on the user's desk
  extend instance objchildren {
    attrnotallowed<attrallclass, attrallcomponent> = {
      {Order, APPDUCXSAMPLE@200.200},
      {Invoice, APPDUCXSAMPLE@200.200}
    }
  }
}
```

5.8 Software products, software editions and software solutions

Software products, software editions and software solutions are defined as instances within a Fabasoft app.ducx project. To export a software edition or software solution open the “File” menu, click “Export” and click “Fabasoft app.ducx” > “Extract Solution”. Click Next. Select a project that contains a software edition or software solution, specify a folder and click “Finish”. All files of the software edition or software solution are exported as a single ZIP file. Alternatively an Apache Ant task can be used (see chapter 3.5 “Build and test environments”).

Note: Software products and software editions are for Fabasoft internal use only. Software solutions are designed for packaging software projects of Fabasoft partners and customers.

Cloud profile note: The definition of software edition and solution instances is forbidden.

Available properties can be found in the following chapters.

5.8.1 Software product

- `COOSYSTEM@1.1:prodname`
  Defines the name of the software product.
- `COOSYSTEM@1.1:prodverscode`
  Defines the version of the software product.
- `COOSYSTEM@1.1:prodstate`
  Defines whether the software product is in development (`PRODST_DEVELOP`) or is in production (`PRODST_INSTALLED`).
- `COOSYSTEM@1.1:prodcopyright`
  Defines the copyright of the software product.
- `COOSYSTEM@1.1:prodcopyrightbmp`
  Defines the copyright image of the software product.
- `COOSYSTEM@1.1:prodcomponents`
  Defines references of software components that are contained in the software product.
- `COOSYSTEM@1.1:proddemocomponents`
  Defines references of demo software components that are contained in the software product.
- `COOSYSTEM@1.1:produnselcomponents`
  Defines references of software components that are not installed by default when installing the software product.
- `COOSYSTEM@1.1:prodexcomponents`
  Defines references of software components that are not contained any longer in the software product.
- `COOSYSTEM@1.1:prodadminfiles`
  Contains administrative COO files.
- `COOSYSTEM@1.1:prodconfigexpr`
  Contains app.ducx Expression Language to customize the Fabasoft Folio Domain.
- `COOSYSTEM@1.1:compcontents`
  Contains content files like help files.

5.8.2 Software edition

- `COOSYSTEM@1.1:prodbaseprodobjs`
  Contains the software products the software edition is based on.
- `COOSYSTEM@1.1:editallowedbaseeditions`
  Contains the references of allowed base software editions for this software edition. At least one

of the allowed base software editions must be installed in the Fabasoft Folio Domain to be able to install this software edition. Leave this property empty, if the software edition should be a stand-alone edition instead of an add-on edition. Fabasoft Folio Compliance is a typical stand-alone edition and Fabasoft app.ducx is a typical add-on edition. Keep in mind that only one stand-alone software edition or software solution can be installed in a Fabasoft Folio Domain.

- `COOSYSTEM@1.1:proddemodata`
  Contains the references of demo data objects.

Note: Additionally, properties of the software product are also available in a software edition.

5.8.3 Software solution

- `COOSYSTEM@1.1:prodbaseprodobjs`
  Contains the software products the software solution is based on.
- `COOSYSTEM@1.1:solallowedbaseeditions`
  Contains the references of allowed base software editions for this software solution. At least one of the allowed base software editions must be installed in the Fabasoft Folio Domain to be able to install this software solution. Leave this property empty, if the software solution should be a stand-alone solution instead of an add-on solution. Keep in mind that only one stand-alone software edition or software solution can be installed in a Fabasoft Folio Domain.
- `COOSYSTEM@1.1:proddemodata`
  Contains the references of demo data objects.

Note: Additionally, properties of the software product are also available in a software solution.

5.8.4 Example

A solution may look like the following example.

Example

```
instance SoftwareSolution SolutionSample {
  prodname = "Sample Solution";
  prodverscode = 1005;
  prodstate = PRODST_INSTALLED;
  prodcopyright = file("resources/copyright.txt");
  prodcopyrightbmp = file("resources/copyright.bmp");
  prodadminfiles<ncname, nccont> = {
    { "baseaclcfg.coo",  file("resources/config/baseaclcfg.coo") },
    { "baseguicfg.coo",  file("resources/config/baseguicfg.coo") },
    { "baseseccfg.coo",  file("resources/config/baseseccfg.coo") }
  }
  proddemodata = "APPDUCXSAMPLE@200.200:DemoDataForSampleSolution";
}
instance DemoData DemoDataForSampleSolution {
  ddcustomizingexprs<ddename, ddeexpr> = {
    { "FolioDataExpression.exp",
      file("resources/imports/FolioDataExpression.exp")
    }
  }
  ddimports<ddidatasource, component> = {
    {
      file("resources/imports/FOLIOIMPORT_00_Group.csv"),
      APPDUCXSAMPLE@200.200
    }
  }
}
```

6 app.ducx Resource Language

The Fabasoft app.ducx resource language of Fabasoft app.ducx allows you to define resources such as string objects, error messages and symbols. Using the app.ducx resource language, you can create culture- and language-independent solutions as it allows you to avoid hard-coded, culture- and language-specific values in your solution.

A resource model block consists of import declarations and resource model elements. The `resources` keyword denotes a resource model block. It must be followed by the reference of your software component and curly braces.

Resource model blocks can only be contained in files with a `.ducx-rs` extension.

Syntax

```
resources softwarecomponent
{
  // Import declarations
  import softwarecomponent;
  // Resource model elements (symbols, strings, error messages)
  ...
}
```

6.1 Defining strings

In Fabasoft Folio, multilingual strings can be stored in instances of object class *String* (`COOSYSTEM@1.1:String`).

Syntax

```
string reference;
```

The `string` keyword is used to define a string object. The actual text that is stored in the string object is defined like any other multilingual string in Fabasoft app.ducx. Please refer to chapter 4.2.4 “Multilingual strings” for more information on how to enter multilingual strings in Fabasoft app.ducx.

The main purpose of string objects is to be used for defining the label text of *branches* in *dialogs* of virtual applications.

Most dialogs, for instance, have a *Cancel* branch for aborting the virtual application. In order to avoid having to enter the string “Cancel” for each *Cancel* branch, you simply define a string object holding the string “Cancel”, and reuse it for each *Cancel* branch.

For detailed information on virtual applications, please refer to chapter 8.3 “Defining a virtual application”.

Example

```
resources APPDUCXSAMPLE@200.200 {
  import COOSYSTEM@1.1;
  // String object for storing the string "Print Invoice" which is to be
  // used for a branch in a dialog for editing invoice objects
  string StrPrintInvoice;
}
```

6.2 Defining error messages

Syntax

```
errormsg reference;
```

The `errormsg` keyword is used to define a custom error message that you can use when raising exceptions. It must be followed by the reference and a semicolon.

Example

```
resources APPDUCXSAMPLE@200.200 {
  import COOSYSTEM@1.1;
  errormsg NoInvoiceFound;
}
```

6.3 Defining symbols

Syntax

```
symbol SymbolApprove {
  images<symbolimageformat, content> = {
    {
      SF_VALUE,
      file("filename")
    }
  }
}
```

The `symbol` keyword is used to define a symbol. It must be followed by the reference and a block referencing one or more symbol files of different formats and sizes, distinguished by the enumeration property `COODESK@1.1:symbolimageformat`.

It is suggested to import symbol files to the folder `resources/symbols` of your app.ducx project. In the example, the binary content of the symbol files referenced in the symbol definition is retrieved from the relative path `resources/symbols`.

Symbols can be referenced by object classes, branches, toolbar buttons, form pages and access types.

The enumeration items `SF_BMP16` and `SF_GIF16` are available for backward compatibility and should not be used any longer. Define at least a 16 x 16 pixels PNG symbol for the enumeration item `SF_PNG16`. In some cases bigger symbols are displayed in the user interface. For these cases the enumeration items `SF_PNG20`, `SF_PNG24`, `SF_PNG256` and `SF_PNG512` are available. If no symbol of the required size is defined, a fallback symbol will be displayed in the user interface.

In this example, two symbols are defined. The symbol files must be imported to the corresponding folder in the Fabasoft app.ducx project.

Example

```
resources APPDUCXSAMPLE@200.200 {
  import COOSYSTEM@1.1;
  import COODESK@1.1;
  import COOACLEDIT@1.1;
  symbol SymbolPrintInvoice {
    images<symbolimageformat, content> = {
      {
        SF_PNG16,
        file("resources/symbols/PrintInvoice16.png")
      },
      {
        SF_PNG512,
        file("resources/symbols/PrintInvoice512.png")
      }
```

```
    }
  }
  symbol SymbolApprove {
    images<symbolimageformat, content> = {
      {
        SF_PNG16,
        file("resources/symbols/Approve16.png")
      }
    }
  }
}
```

7 app.ducx User Interface Language

The Fabasoft app.ducx user interface language allows you to define forms, form pages, menu items and other user interface elements for your object classes.

A user interface model block consists of import declarations and user interface model elements. The `userinterface` keyword denotes a user interface model block. It must be followed by the reference of your software component and curly braces.

User interface model blocks can only be contained in files with a `.ducx-ui` extension.

Syntax

```
userinterface softwarecomponent
{
  // Import declarations
  import softwarecomponent;
  // User interface model elements (forms, form pages, ...)
  ...
}
```

7.1 Defining forms and form pages

Fabasoft Folio distinguishes two types of forms:

- Forms for editing and searching are displayed by the attribute editor when the *Edit Properties*, *Read Properties* or *Find* menu item is invoked by the user. Forms for editing and searching are usually simply referred to as “forms”.
- A desk form is displayed on the right-hand side of the client when an instance of a compound object class is selected in the tree view or when the *Explore* menu item is invoked by the user.

7.1.1 Defining a form

Syntax

```
form reference {
  audience = audience;
  // New form pages can be defined inline
  formpage reference {
    audience = audience;
    dataset {
      class;
      property;
      ...
    }
  }
  // Existing form pages can be reused by adding their references to the
  // form block in the order they should appear on the form
  reference;
```

```
}
```

A form consists of form pages holding the properties that should be displayed. Form pages can either be defined inline or existing form pages can be reused.

The `form` keyword is used to define a form. It must be followed by a reference and curly braces.

7.1.1.1 Defining the audience for a form

The `audience` keyword is used to define the target audience of a form. The next table shows the possible values that can be assigned to `audience`. When `audience` is not specified implicitly `enduser` is the target audience.

Audience	Description
`enduser`	The form is usable by all users. The ACL `COOSYSTEM@1.1:DefaultAdministratorACL` is assigned to the form.
`administrator`	The form is usable by administrators only. The ACL `COOSYSTEM@1.1:DefaultDeveloperACL` is assigned to the form.

Table 11: Audience of a form or form page

7.1.1.2 Defining the formcaption for a form

The property `formcaption` stores the multilingual caption of the display item. It stores one name for each supported language. An entry in the multilingual files is only generated if explicitly used.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  form FormInvoice {
    formcaption = {}
  }
}
```

7.1.1.3 Adding form pages to a form

Each form can consist of multiple form pages that can either be reused or defined inside the `form` block.

The `formpage` keyword is used to define a new form page inside the `form` block. For reusing an existing form page, you just need to provide the reference of the form page followed by a semicolon.

The order of the form pages in the `form` block also determines their order on the form when it is displayed in the user interface.

7.1.1.4 Enabling the generic view

When enabling the generic view, properties that are not explicitly assigned to a form page are displayed on a system-generated form page. The name of the system-generated form page corresponds to the name of the object class the properties belong to.

To enable the generic view for a form, `COOATTREDIT@1.1:formgeneric` can be set to `true`. In this case, it is not necessary to define any form pages as all properties of the object class of the current

object are displayed in a generic manner. In the generic view, the order of the properties on the system-generated form page is determined by the object class definition.

7.1.1.5 Inheriting form pages from base classes

When setting `inherit` to `true`, the form pages inherited from the base classes are displayed after the form pages defined in the current form.

When setting `inheritfrom` to a class, the form pages inherited from this class and its base classes are displayed. This setting is evaluated after the property `inherit`.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  form FormInvoice {
    audience = enduser;
    inherit = true;
    inheritfrom = CompoundObject;
    formpage PageInvoiceGeneral {
      audience = enduser;
      symbol = SymbolInvoice;
      dataset {
        invoicestate;
        invoiceorder;
      }
    }
    formpage PageInvoiceDates {
      audience = enduser;
      symbol = SymbolInvoiceDates;
      dataset {
        invoicedate;
        invoicepaymentdate;
      }
    }
  }
}
```

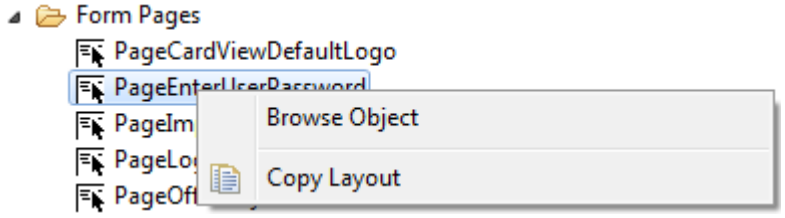
7.1.2 Defining a form page

A form page is defined in a `formpage` block that must be nested within a `form` block. The `formpage` keyword must be followed by the reference of the form page and curly braces.

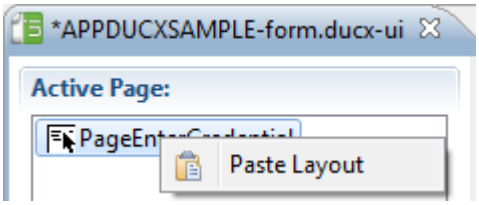
A form page consists of a `dataset` block holding the properties and classes which can be used in the layout definitionto be displayed, an optional layout definition, and some general settings concerning the symbol displayed on the tab and the target audience.

7.1.2.1 Reuse an existing form page

If an existing form page should be reused open the Software Component References folder and select Copy Layout from the context menu of a form page. If the selected form page has no XML layout an error message is shown.



To insert the layout of the selected form page, switch to the form designer and select Paste Layout in the context menu of the destination form page.



7.1.2.2 Assigning a symbol to a form page

The `symbol` keyword is used to assign a symbol to a form page. The symbol referenced by the `symbol` assignment is displayed on the tab of the form page.

Note: The Fabasoft app.ducx resource language must be used for defining custom symbols. However, you can also reuse existing symbols. For further information on defining symbols, refer to chapter 6.3 “Defining symbols”.

7.1.2.3 Defining the audience for a form page

As is the case for a form, the `audience` keyword is used to define the target audience of a form page. For the possible values that can be assigned to `audience` see chapter 7.1.1.1 “Defining the audience for a form”.

Please note that when a form is presented to an end-user that contains a form page with `audience` set to `administrator`, the form page in question is not displayed.

7.1.2.4 Adding properties to a form page

The `dataset` keyword is used to assign properties or fields to a form page.

All properties displayed on a form page must be listed in the `dataset` block nested within the `formpage` block. Semicolons must be used to separate multiple entries.

If a class has been added to the `dataset` block, all defined properties of only this class can be used in the layout. Inherited properties have to be added explicitly.

7.1.2.5 Adding fields to a form page

Fabasoft app.ducx also allows you to reference fields in the `dataset` block nested within the `formpage` block. You may only use fields that have been defined in a `fields` block using the Fabasoft app.ducx object model language, and each field referenced in the `dataset` block must be preceded by the keyword `field`.

Fields may be utilized to temporarily compute values that should not be stored persistently.

The optional `get` and `set` keywords allow you to define get and set expressions for calculating the field value. The get expression is evaluated before a field is displayed. The set expression is calculated when leaving the form page displaying a field or when clicking a branch (e.g. *Apply*).

When a get or set expression of a field is evaluated, the local scope `this` contains the vApp state dictionary. The global scope `::this` contains a dictionary holding the keys listed in the next table.

Key	Description
`roots`	The `roots` and `root` keys both store the current object.

`root`	
`value`	The `value` key stores the current value of the field. **Note:** This key is only available for set expressions.
`attribute`	The `attribute` key stores the object representing the field.

Table 12: Global scope for get and set expressions of fields

7.1.2.6 Laying out properties on a form page

A `formpage` block can contain an optional `layout` block that allows you to define a specific layout of controls on the form page.

The `layout` block consists of `row` blocks. A row may have several columns. A standard control consists of four columns, one column for the label and three columns to display the data. You can use the keywords listed in the next table to influence the way the property is displayed and behaves.

Keyword	Description
`rowspan`	The `rowspan` keyword is used to define how many rows the control spans.
`colspan`	The `colspan` keyword is used to define how many columns the control spans (default is four; one column for the label and three columns for the data).
`valign`	The `valign` keyword is used to define the vertical alignment of text within the control (`top`, `middle` or `bottom`).
`halign`	The `halign` keyword is used to define the horizontal alignment of text within the control (`left`, `center` or `right`).
`fontfamily`	The `fontfamily` keyword is used to specify the name of a font family used within a control with a CSS conformant string, i.e.: `fontfamily = "verdana, sans-serif"`. **Note:** Some controls override the used font, so that this element is ignored. This element is available in Version 2013 Spring Release or later.
`fontcolor`	The `fontcolor` keyword is used to define the color of text within the control (hexadecimal value).
`fontsize`	The `fontsize` keyword is used to define the size of text within the control (pt, px, em).
`labeltext`	The `labeltext` keyword is used to reference a string object that will provide the text of the label.
`labelvisible`	The `labelvisible` keyword is used to define whether the label is visible (`true` or `false`).
`labelrowspan`	The `labelrowspan` keyword is used to define how many rows the label spans.
`labelcolspan`	The `labelcolspan` keyword is used to define how many columns the label spans.

`labelposition`	The `labelposition` keyword is used to define the position of the label (`top`, `bottom`, `left` or `right`).
`labelvalign`	The `labelvalign` keyword is used to define the vertical alignment of a label (`top`, `middle` or `bottom`).
`labelhalign`	The `labelhalign` keyword is used to define the horizontal alignment of a label (`left`, `center` or `right`).
`labelfontfamily`	The `labelfontfamily` keyword is used to specify the name of a font family used for the label text of a control with a CSS conformant string, i.e.: `fontfamily = "verdana, sans-serif"`. **Note:** This element is available in Version 2013 Spring Release or later
`labelfontcolor`	The `labelfontcolor` keyword is used to define the color of the label text (hexadecimal value).
`labelfontsize`	The `labelfontsize` keyword is used to define the size of the label text (pt, px, em).
`detail`	The `detail` keyword is used to define the columns of an aggregate. Usage: `detail = layout {...}`
`simple`	The `simple` keyword is used to define the columns of an aggregate that should be displayed in the simple view for in-place editing. Usage: `simple = layout {...}`
`empty`	The `empty` keyword is used to define a cell in the grid layout without content. Usage: `empty;` or `empty { colspan = ...; }`
`mustbedef`	The property must contain a value, if the expression returns `true`. Usage: `mustbedef = expression {...}`
`changeable`	The property is changeable in the user interface, if the expression returns `true`. Usage: `changeable = expression {...}`
`validate`	The expression is used to check whether the value entered in the property is valid. Usage: `validate = expression {...}`
`change`	If the value of the property gets changed, the expression is executed. Usage: `change = expression {...}`
`searchchange`	If the value of the property gets changed in context of a search, the expression is executed. Usage: `searchchange = expression {...}`
`selchange`	If an object of the property gets selected, the expression is executed. Usage: `selchange = expression {...}`
`visible`	The property is only visible in the user interface, if the expression returns `true`. If a row does not contain any visible property, the whole row is

	removed. Usage: `visible = expression {...}`
`controlstyle`	Style information for a control. Either a list of control styles (enumeration type `COOATTREDIT@1.1:ControlStyle`) or a control style definition (object of object class `COOATTREDIT@1.1:ControlStyleDefinition`) is provided. A control style definition references specific control styles. The following control styles are supported: • `CTRLSTYLE_DISPLAYREFERENCE`: display reference of languages, software components, software products and component objects in an object pointer property • `CTRLSTYLE_DISABLECONTEXTMENU`: disable context menu on objects in object pointer property • `CTRLSTYLE_DISABLECREATE`: disable create button for single object pointer property • `CTRLSTYLE_DISABLESEARCH`: disable search button for single object pointer property • `CTRLSTYLE_DISABLEQUICKSEARCH`: disable quick search field for object list property • `CTRLSTYLE_DISABLEINPLACECHANGE`: disable F2 inplace editing in an aggregate property • `CTRLSTYLE_DISABLEDRAGTARGET` • `CTRLSTYLE_AUTONUMERATE` • `CTRLSTYLE_IGNOREDBYTREE` Usage: `controlstyle = expression {...}`
`controloptions`	Additional options for a control as a string. Please refer to [Faba10c] for a description of possible options. Usage: `controloptions = expression {...}`
`sort`	The `sort` property sorts values in a column of a list. Available modes are `up` and `down`.
`fixed`	The `fixed` property allows to make a column not horizontal scrollable. There are at least three columns necessary and it will not work on the last two columns. The column index starts with 1.
`group`	The `group` property allows to group values in a column of a list. Available modes are `unique`, `alpha`, `year`, `quarter, month, week, day` and `hour`.
`index`	The order of grouped and sorted columns is defined by `index`.
`aggregation`	The `aggregation` property is used to apply a function on a column. Predefined values in Fabasoft Folio are `MenuSum`, `MenuAverage` or `MenuCount`.
`weight`	The value of the `weight` property is an expression which has to return one of the enumeration values defined in `FSCVAPP@1.1001:HighlightType`. The weight of a form page element is used to draw the users attention to this particular element.

accset	The accset property is an expression which returns a COOSYSTEM@1.1:AccessType. The current user must have access to the current object via this AccessType (based on the ACL evaluation of the Fabasoft Folio Kernel) in order to change the value of this property.
accsetline	The accsetline property is an expression which returns a COOSYSTEM@1.1:AccessType. This expression is evaluated for each line in an aggregate list, so that special access rights can be applied for each line in the list.
width	The width property is used to define the width of a column. There is no measure used. App.ducx removes added measures automatically.
height	The height property is used to define the height of a column. There is no measure used. App.ducx removes added measures automatically. The height of a row in a detail layout cannot be changed.

Table 13: Elements allowed within a layout block

In a row block, the reference of a property or field can be preceded by the reference of a control. The control determines the behavior as well as the look and feel of this property or field.

Table 14 provides a list of available controls. For detailed information on controls and the associated parameters, please refer to [Faba10c].

Control	Description
COOATTREDIT@1.1:CTRLAcl	This control is used to display and edit access control lists.
COOATTREDIT@1.1:CTRLAttrpath	This control is used to display an property path.
COOATTREDIT@1.1:CTRLBase	This control is used for simple data types like integer, float, string and object pointers.
COOATTREDIT@1.1:CTRLCont	This control is used to upload content.
COOATTREDIT@1.1:CTRLCurr	This control is used to display and edit currency properties.
COOATTREDIT@1.1:CTRLDateTime	This control is used to display date and datetime properties.
COOATTREDIT@1.1:CTRLMLName	This control is used to display and edit multilingual strings (using the compound type COOSYSTEM@1.1:LanguageStringList).
COOATTREDIT@1.1:CTRLObjNav	This control is used to display and edit an object pointer describing a hierarchical relationship.
COOATTREDIT@1.1:CTRLPassword	This control is used for entering passwords.
COOATTREDIT@1.1:CTRLPhone	This control is used to display and edit telephone numbers.
COOATTREDIT@1.1:CTRLPick	This control is used to display a list of enumeration entries

	or a list of objects as a group of radio buttons or checkboxes.
`FSCVENV@1.1001:CTRLRating`	This control is used to display a star rating.
`COOATTREDIT@1.1:CTRLResearch`	This control is used to display the input field for a research term.
`COOATTREDIT@1.1:CTRLStrList`	This control is used to display a list of strings in a combo box.
`COOATTREDIT@1.1:CTRLText`	This control is used to enable multiline text input for string lists.
`COOATTREDIT@1.1:CTRLTimespan`	This control is used to display and edit a time span in days, hours, minutes and seconds.
`COOATTREDIT@1.1:CTRLTimestamp`	This control is used to display and edit a time stamp in hours, minutes and seconds.
`COOATTREDIT@1.1:CTRLTree`	This control is used to display the tree.
`COOATTREDIT@1.1:CTRLViewCont`	This control is used to display content.
`COOATTREDIT@1.1:CTRLXMLAttr`	This control is used to display attributes of an XML document.
`FSCDOX@1.1001:CTRLHtmlEditor`	This control is used to attach an HTML editor to a string, string list, content or content aggregate.
`FSCSIMLIST@1.1001:CTRLSimList`	This control is used to display a simple object list, comparable to the Microsoft Outlook mail-add field.
`FSCVENV@1.1001:CTRLDictionary`	This control is used to display dictionaries.
`FSCVENV@1.1001:CTRLInstantiate`	This control is used to display the creatable objects and templates in the dialog box displayed when creating a new object.
`FSCVENV@1.1001:CTRLPict`	This control is used to display and upload pictures.
`FSCVENV@1.1001:CTRLProgress`	This control is used to display a progress bar.
`FSCVENV@1.1001:CTRLRawText`	This control is used to transfer the contents of script component objects to the web browser and can be used for injecting HTML code into the current page.
`FSCVENV@1.1001:CTRLURLNav`	This control is used to display and edit a URL and can be attached to a `string` property.

Table 14: Predefined controls in Fabasoft Folio

The following example demonstrates how to use the form page layout elements discussed in this chapter.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  form OrderForm {
    audience = enduser;
    inherit = true;
    formpage OrderGeneralPage {
      audience = enduser;
      symbol = SymbolOrder;
      dataset {
        orderstate;
        orderinvoice;
        orderdate;
        orderapproved;
        field orderagent {
          get = expression {
            coouser;
          }
        }
        field ordertotal;
      }
      layout {
        // APPDUCXSAMPLE@200.200:orderstate and APPDUCXSAMPLE@200.200:ordertotal
        // should be displayed side-by-side
        row {
          APPDUCXSAMPLE@200.200:orderdate {
            colspan = 4;
            labelposition = left;
            labeltext = StrOrderdateLabel;
          }
          APPDUCXSAMPLE@200.200:ordertotal {
            colspan = 4;
            labelposition = left;
          }
        }
        // APPDUCXSAMPLE@200.200:orderdate spans two rows, uses as many
        // columns as orderstate and ordertotal together and no label
        // is displayed
        row {
          APPDUCXSAMPLE@200.200:orderdate {
            rowspan = 2;
            colspan = 8;
            labelvisible = false;
          }
        }
        row {
        }
        // APPDUCXSAMPLE@200.200:orderapproved uses the control
        // COOATTREDIT@1.1:CTRLBase in order to be displayed as a group
        // of radio buttons
        row {
          COOATTREDIT@1.1:CTRLBase("UseRadio=true") orderapproved {
            colspan = 8;
            labelposition = left;
          }
        }
        row {
          orderagent {
            colspan = 8;
            labelposition = left;
          }
        }
      }
    }
    formpage OrderPositionsPage {
      audience = enduser;
      symbol = SymbolOrderPositions;
      dataset {
```

```
        user;
        orderpositions;
      }
    }
  }
}
```

The following example demonstrates how to define a form page and apply the `COOATTREDIT@1.1:CTRLObjNav` control to the `FSCFOLIO@1.1001:orgindustry` property of an organization to allow the user to select an industry from a hierarchical list of industries.

Note: In Fabasoft Folio, industries are represented as instances of object class `FSCFOLIO@1.1001:FunctionTerm`. The root object for the available industries is `FSCFOLIO@1.1001:TermIndustry`, where the top-level industries must be referenced in the `FSCTERM@1.1001:narrowercompterms` property. Subordinated industries must be referenced in the `FSCTERM@1.1001:narrowercompterms` property of the `FSCFOLIO@1.1001:FunctionTerm` object describing an industry.

The control parameters `Root` and `Lists` of the `COOATTREDIT@1.1:CTRLObjNav` control must be initialized with the addresses of the corresponding expression objects responsible for determining the root object of the hierarchy and the lists describing the hierarchy. To find out the address of the expression objects, refer to the “Address Assignment” list in the Eclipse Project Explorer.

For detailed information on the control parameters of the `COOATTREDIT@1.1:CTRLObjNav` control refer to [Faba10c].

Example

app.ducx Object Model Language

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCFOLIO@1.1001;
  import FSCTERM@1.1001;
  // Assuming that APPDUCXSAMPLE@200.200:ExprTermRoot is assigned the address
  // COO.200.200.1.100 in the address assignment list
  instance Expression ExprTermRoot {
    exprtext = expression {
      #TermIndustry;
    }
  }
  // Assuming that APPDUCXSAMPLE@200.200:ExprTermList is assigned the address
  // COO.200.200.1.101 in the address assignment list
  instance Expression ExprTermList {
    exprtext = expression {
      dictionary @list = {
        cls = #TermComponentObject,
        attrs = #narrowercompterms
      };
      return dictionary @treelist = {
        destattr = #orgindustry,
        lists = @list
      };
    }
  }
}
```

app.ducx User Interface Language

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCFOLIO@1.1001;
  form FormOrgIndustry {
    audience = enduser;
```

```
    inherit = false;
    formpage PageOrgIndustry {
      audience = enduser;
      dataset {
        orgindustry;
      }
      layout {
        row {
          // Assuming that APPDUCXSAMPLE@200.200:ExprTermRoot is assigned the address
          // COO.200.200.1.100 and APPDUCXSAMPLE@200.200:ExprTermList is assigned the
address
          // COO.200.200.1.101 in the address assignment list
          COOATTREDIT@1.1:CTRLObjNav("Root=COO.200.200.1.100 Lists=COO.200.200.1.101
            IgnoreCompoundFlag=true") FSCFOLIO@1.1001:orgindustry {
            colspan = 4;
            labelposition = left;
          }
        }
      }
    }
  }
}
```

7.1.3 Using the form designer

The layout can be defined using the provided graphical form designer. You can switch from the *Code* pane to the *Form Pages* pane. The *Palette* contains all properties that are defined in the `dataset`.

The form designer provides following features:

- The properties can be adjusted within the form page by drag-and-drop.
- Labels and fields can be spanned horizontally or vertically over multiple columns or lines.
- Pressing the `Alt` key allows to select the label and to adjust it within the field. A label can be positioned left, right, at the top or at the bottom of a field.
- A horizontal rule can be inserted by selecting it from the “Static Controls”.
- To filter the available component objects use the **Filter** field. Clicking “x” deletes the filter.
- Use the context menu to define a specific control for a property.

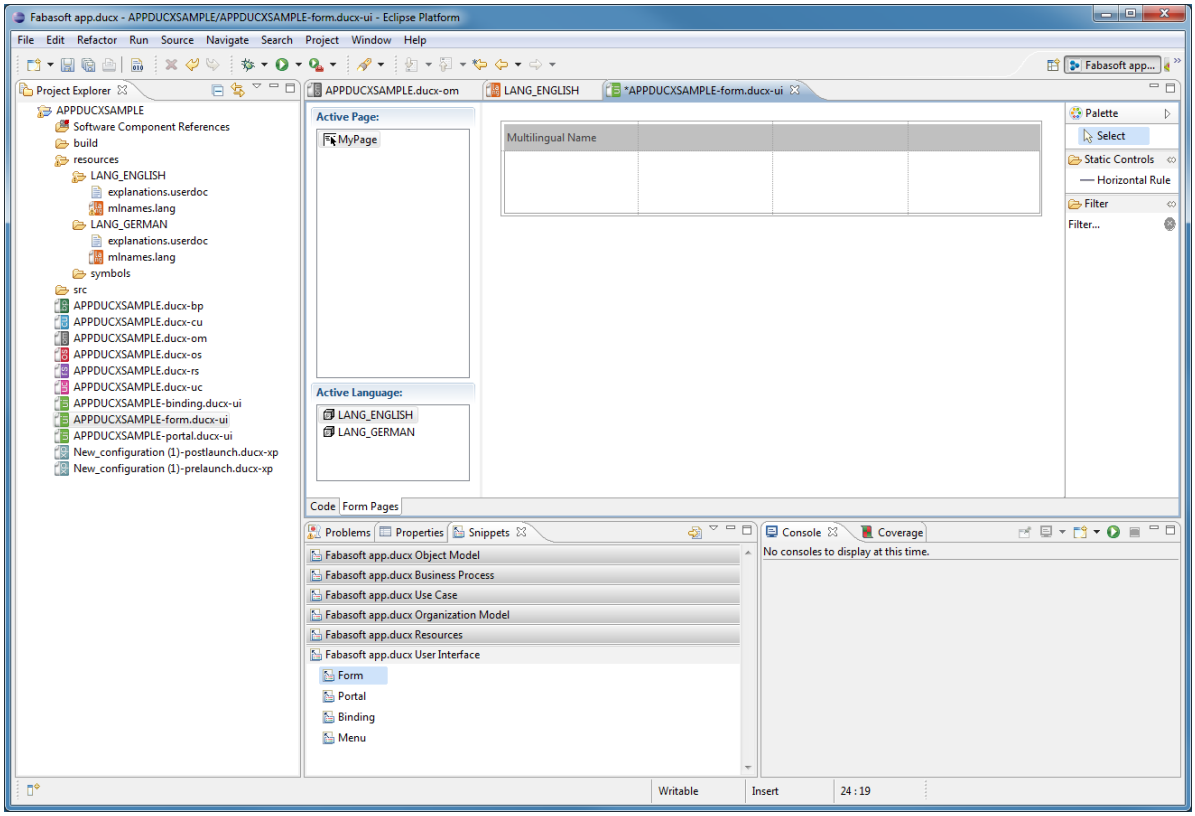


Figure 18: Example of a form page

7.1.4 Defining a desk form

Syntax

```
deskform reference {
  audience = audience;
  dataset {
    property;
    ...
  }
}
```

A desk form is a special kind of form that is used for displaying object list properties of compound objects in the so called explore view.

For instance, a desk form is used for displaying the object list properties shown in the list view on the right-hand side when you select an instance of object class *Person* in the tree view of the desk.

The `deskform` keyword is used to define a desk form. It must be followed by a reference and curly braces.

7.1.4.1 Defining the audience for a desk form

Inside a `deskform` block, the `audience` keyword can be used for defining the audience for a desk form. For further information concerning the `audience` keyword please refer to chapter 7.1.2.3 “Defining the audience for a form page”.

7.1.4.2 Adding properties to a desk form

All object list properties displayed on a desk form must be listed in a `dataset` block nested within the `deskform` block. Semicolons must be used to separate multiple entries.

The order of the properties in the `dataset` block also determines their order on the desk form.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  deskform CustomerDeskForm {
    audience = enduser;
    dataset {
      customerorders;
    }
  }
}
```

7.2 Extending existing desk forms, forms and form pages

Fabasoft app.ducx allows you to extend existing forms with new form pages as well as to add properties to existing desk forms and form pages belonging to other software components.

Cloud profile note: The extension of desk forms, forms and form pages that belong to another non-friend software component is discouraged.

7.2.1 Extending an existing desk form

Syntax

```
extend deskform reference {
  dataset {
    property;
    ...
  }
}
```

With the `extend deskform` keywords, you can add properties to a desk form that is part of another software component.

7.2.2 Extending an existing form

Syntax

```
extend form reference {
  // New form pages can be defined inline
  formpage reference {
    audience = audience;
    dataset {
      property;
      ...
    }
  }
  // Existing form pages can be reused
  reference;
}
```

A form of another software component may be extended with form pages belonging to your software component.

To extend a form that is part of another software component, you can use the `extend form` keywords followed by the reference of the form you want to extend, and curly braces. You may either define new form pages inside the extension block or reference existing form pages of your software component.

New form pages added in an extension block are appended to the form after the existing form pages.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCFOLIOGUI@1.1001;
  // Extend form FSCFOLIOGUI@1.1001:FormPersonUV with a new form page
  // APPDUCXSAMPLE@200.200:PagePersonOrders
  extend form FormPersonUV {
    formpage PagePersonOrders {
      audience = enduser;
      symbol = SymbolOrder;
      dataset {
        customerorders;
      }
    }
  }
}
```

7.2.3 Extending an existing form page

Syntax

```
extend formpage reference {
  dataset {
    property;
    ...
  }
}
```

With the `extend formpage` keywords, you can add properties to a form page that is part of another software component.

Note: Properties added to a form page must also be part of the corresponding object class in order to be displayed on the form page.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCFOLIOGUI@1.1001;
  // Extend form page FSCFOLIOGUI@1.1001:PagePerson with a new
  // property APPDUCXSAMPLE@200.200:customertype
  extend formpage PagePerson {
    dataset {
      customertype;
    }
  }
}
```

7.3 Defining menus, button bars and task panes

With the Fabasoft app.ducx user interface language of Fabasoft app.ducx, you can define the following user interface elements in addition to forms and form pages:

- menu items and menu bars
- button bars and buttons
- task panes

7.3.1 Defining a menu

Fabasoft Folio distinguishes between two types of menus:

- Menu items are mainly used for invoking use cases, but can also contain one or more menu items in order to allow the definition of menu hierarchies with sub-menus.
- A menu root is required as the outermost container holding one or more menu items.

7.3.1.1 Defining a new menu item

Usually, menu items are simply referred to as menus. A menu can either reference a use case that is invoked when the user selects the menu item in the user interface, or it can contain one or more sub-menus to create a menu hierarchy.

The `menu` keyword is used to define a menu. It must be followed by the reference and curly braces.

Using the `accel` keyword, an accelerator object can be assigned to the menu to allow the user to use a shortcut for accessing the menu.

The alias group is used to insert a seperator before the menu entry.

7.3.1.1.1 Defining a menu for invoking a use case

Syntax

```
menu reference {
  group = booleanvalue;
  accel = accelerator;
  usecase = usecase;
}
```

The use case that should be invoked when the menu is selected by the user must be referenced using the `usecase` keyword.

Please note that the app.ducx compiler automatically generates a menu item for use cases that are defined using the `menu usecase` keywords in the app.ducx use case language (for further information refer to chapter 8.2.2 “Defining a new menu use case”). Thus, there usually is no need to define a custom menu when using the `menu usecase` keywords to define use cases that should be invoked from a menu.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COODESK@1.1;
  menu MenuMarkAsShipped {
    group = true;
    accel = AccelCtrlO;
    usecase = MarkOrderAsShipped;
```

```
  }
}
```

7.3.1.1.2 Defining sub-menus

Syntax

```
menu reference {
  accel = accelerator;
  entries = {
    menu,
    ...
  }
}
```

The `entries` keyword is used to create a hierarchical menu containing one or more sub-menus, which must be menu objects themselves. Multiple entries must be separated by commas.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  menu MenuSetOrderState {
    entries = {
      MarkOrderAsShipped,
      MarkOrderAsCompleted
    }
  }
}
```

7.3.1.2 Defining a new menu root

Syntax

```
menuroot reference {
  entries = {
    menu,
    ...
  }
}
```

A menu root is required as the starting point for context menus and menu bars.

The `menuroot` keyword is used to define a menu root. It must be followed by the reference and curly braces. In the `menuroot` block, the `entries` keyword is used for referencing the menu items.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COODESK@1.1;
  menuroot MenuRootOrderContext {
    entries = {
      MenuObjectUnshare,
      MenuSeparator,
      MenuObjectRead,
      MenuObjectEdit,
      MenuSeparator2,
      MenuSetOrderState
    }
```

```
  }
}
```

7.3.2 Extending an existing menu

Syntax

```
extend menu reference {
  entries = {
    menu,
    ...
  }
}
```

The `extend menu` keywords are used to add menu items to an existing menu or menu root belonging to another software component.

Note: Menus belonging to another software component can only be extended with menu items belonging to your software component.

Cloud profile note: The extension of menus that belong to another non-friend software component is discouraged.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COODESK@1.1;
  // Add menu APPDUCXSAMPLE@200.200:CreateOrderWizard to the context menu
  // of contact persons
  extend menuroot MenuRootContext {
    entries = {
      CreateOrderWizard
    }
  }
}
```

7.3.3 Weighted menu

Syntax

```
extend menu reference {
  weighted<..> = {
    ...
  }
}
```

Menu entries may be declared to be shown as buttons. In the menu extension block, `weighted` is used to assign menu entries.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COODESK@1.1;
  import COOATTREDIT@1.1;
  // Add menu MenuObjectEdit to the buttons
  // of MenuRoot
  extend menu MenuRoot {
    weighted<weightedmenuentry, component> = {
```

```
      { MenuObjectEdit, APPDUCXSAMPLE@200.200 }
    }
  }
}
```

7.3.4 Defining a task pane

Syntax

```
taskpane reference {
  entries = {
    menu,
    ...
  }
}
```

A task pane is comprised of a list of menu items displayed in the left part of the quick view (see next figure).

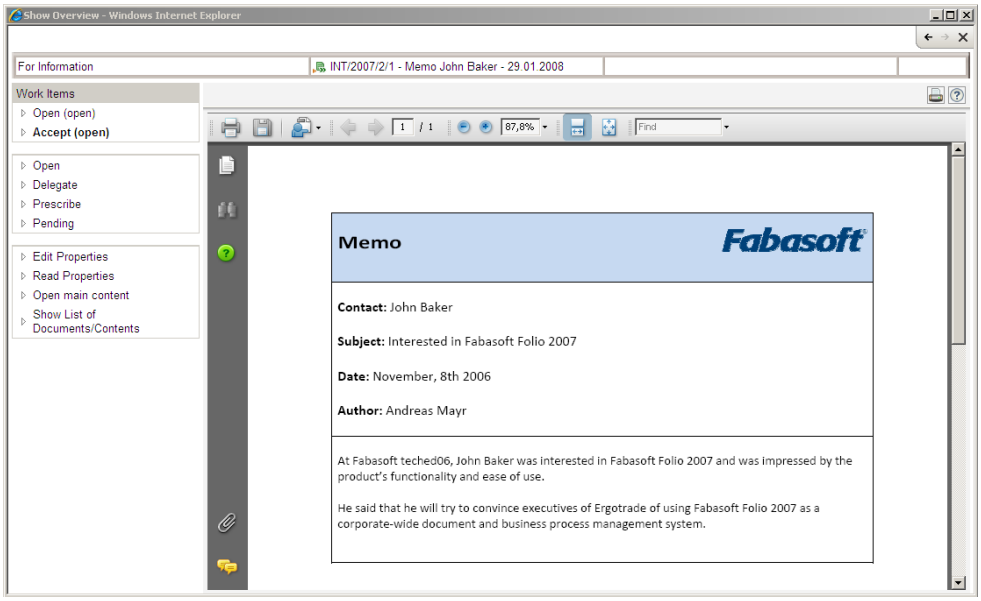


Figure 19: Task panes displayed on the left-hand side of the quick view

The `taskpane` keyword is used to define a task pane. It must be followed by the reference and curly braces.

Nested within the `taskpane` block, the `entries` keyword is used to reference the menu items displayed on the task pane. Multiple entries must be separated by commas.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COODESK@1.1;
  taskpane TaskPaneCustomer {
    entries = {
      MenuObjectEdit,
      MenuMarkOrderAsShipped,
      MenuCreateInvoice
    }
  }
}
```

7.3.5 Extending an existing task pane

Syntax

```
extend taskpane reference {
  entries = {
    menu,
    ...
  }
}
```

The `extend taskpane` keywords are used to add menu items to an existing task pane belonging to another software component.

Note: Task panes belonging to another software component can only be extended with menu items belonging to your software component.

Cloud profile note: The extension task panes that belong to another non-friend software component is discouraged.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COOWF@1.1;
  // Add menu APPDUCXSAMPLE@200.200:MenuRerouteActivity to task pane
  // COOWF@1.1:TaskPaneActivity
  extend taskpane TaskPaneActivity {
    entries = {
      MenuRerouteActivity
    }
  }
}
```

7.4 Assigning user interface elements to an object class

Fabasoft app.ducx allows you to assign forms, symbols, context menus and menu bars, button bars and task panes to object classes. These assignments are referred to as user interface element bindings.

It is recommended that you create a separate `.ducx-ui` file for defining your user interface element bindings such as `bindings.ducx-ui` although you can define all user interface model-related elements within a single `.ducx-ui` file.

For all user interface element bindings, the `extend class` keywords must be used, followed by the reference of the object class to extend and curly braces. All user interface element extensions for an object class can be combined into a single extension block.

Note: Forms, symbols, menus and other user interface elements can be assigned to an object class within a single `extend class` block. However, you must obey the element order as listed in the following example.

Cloud profile note: The assignment of user interface elements to an object class that belongs to another non-friend software component is discouraged.

Syntax

```
extend class reference {
  symbol = ...
  menus {
    ...
```

```
  }
  taskpanes {
    ...
  }
  forms {
    ...
  }
}
```

7.4.1 Assigning a symbol to an object class

Syntax

```
extend class reference {
  symbol = symbol;
}
```

A symbol should be assigned to each object class defined. The `symbol` keyword is used for assigning a symbol to an object class. It can be nested within an `symbols` block.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  extend class Order {
    symbol = SymbolOrder;
  }
}
```

7.4.2 Assigning menus to an object class

Menu bars and context menus are assigned to object classes using a `menus` block that must be nested within an `extend class` block.

Each object class has a default menu binding defining the default main menu, context menu, background menu and the container-independent menu. The `default` keyword is used to denote the default menu binding, which must be nested within the `menus` block.

Additionally, you can assign one or more scope-specific menu bindings to an object class. Any valid component object can be used as scope.

When the menu bindings are evaluated by the Fabasoft Folio Kernel, the currently active scope is determined by the expression defined in the *User Interface Scoping Rule* (`COODESK@1.1:UIScopingRule`) referenced in the *Current Domain* (`COOSYSTEM@1.1:CurrentDomain`) object of your Fabasoft Folio Domain. The object returned by the expression is used as the current scope.

Cloud profile note: In Fabasoft Folio Cloud, the current team room's type (represented by the corresponding app object) is used as current scope. For example, within a team room of type "CRM", the `FSCFOLIOCLOUDCRM@1.1001:AppCRM` app object is used as current scope.

If the current scope determined by the *User Interface Scoping Rule* matches the scope object referenced in a menu binding, the corresponding menu is displayed. However, if there is no scope-specific menu available in an object class for the current scope, the default menu is displayed instead.

Syntax

```
extend class reference {
  menus {
    default = {
      mainmenu = menuroot;
      ctxmenu = menuroot;
      winctx = menuroot;
      independent = menuroot;
      expansions {
        expansion point {
          target = property;
          condition = expression {...}
          priority = priority;
          entries = {
            menu
          }
        }
      }
    }
    scope = {
      mainmenu = menuroot;
      ctxmenu = menuroot;
      winctx = menuroot;
      independent = menuroot;
      }
    }
  }
}
```

The keywords listed in the next table can be used within a menu binding block for assigning the different types of menus.

Keyword	Description
mainmenu	With the mainmenu keyword, a menu root can be assigned to an object class, which is used as its main menu.
ctxmenu	With the ctxmenu keyword, a menu root can be assigned to an object class, which is used as its context menu.
winctx	With the winctx keyword, a menu root can be assigned to an object class, which is used as its background menu.
independent	With the independent keyword, a menu root can be assigned to an object class, which is used as its container-independent menu.
expansions	With the expansions keyword, a menu can be assigned to a specific menu expansions point. (See next paragraph).

Table 15: Menu binding keywords

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COODESK@1.1;
  extend class Order {
    menus {
      default = {
        mainmenu = MenuRootOrder;
        ctxmenu = MenuRootOrderContext;
        winctx = MenuRootWinContext;
        independent = MenuRootOLEContext;
```

```
        }
      }
    }
  }
```

7.4.3 Assigning menu items to an object class (Expansion Points)

The default menus, taskpanes and the context menu are prepared for extension by using so called **expansion points**. These are dynamic menus which collect their menu items from the object class of the current container resp. the object classes of the currently selected objects, thus removing the need of directly extending the menus. This is the suggested way of assigning new menu commands to objects.

The expansions points are named after the menus where they are inserted and can be used directly as key for the `expansions` block.

The menus or menu usecases in the `entries` block are inserted into the given menu. Additional restrictions for using the menu items can be specified: The alias `target` limits the menu entries to the given property. The alias `condition` has to evaluate to `true` in order to make the menu entry visible.

The expansion points listed in the next table can be used within the `expansions` block.

Expansions point	Description
`COODESK@1.1:MenuBackgroundExpansion`	This expansion point inserts a menu entry in the background area. A symbol is required for empty object lists.
`COODESK@1.1:MenuRootSimpleExpansion`	This is used to add a menu entry to the simple menu view. This menu contains only symbols. The menu entry requires a selection in the tree.
`COODESK@1.1:MenuObjectDirectExpansion`	This expansion point adds a menu entry in the *Object* menu. The menu items are retrieved from the object class of the container object.
`COODESK@1.1:MenuObjectExpansion`	This expansion point is used to insert a menu entry in the *Object* menu. The menu items are retrieved from the object classes of the currently selected objects.
`COODESK@1.1:MenuToolsDirectExpansion`	This expansion point adds a menu entry in the *Tools* menu. The menu items are retrieved from the object class of the container object.
`COODESK@1.1:MenuToolsExpansion`	This expansion point is needed to add a menu entry in the *Tools* menu. The menu items are retrieved from the object classes of the currently selected objects.
`COODESK@1.1:MenuContextDirectExpansion`	This expansion point is used to extend the context menu. The menu items are retrieved from the object class of the container object, which is also the current object if the context menu is displayed in the tree.

COODESK@1.1:MenuContextExpansion	This expansion point is used to extend the context menu. The menu items are retrieved from the object classes of the currently selected objects.
COODESK@1.1:TaskPaneSelectedExpansion	This expansion point is used to insert a task pane entry. The entry is only activated if an object is selected.
COODESK@1.1:TaskPaneExpansion	This expansion point is used to insert a task pane entry. The menu items are retrieved from the object class of the container object.

Table 16: Expansion points

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COODESK@1.1;
  extend class Package {
    menus {
      default = {
        expansions {
          TaskPaneSelectedExpansion {
            target = objchildren;
            condition = expression { !cooobj.isencrypted }
            priority = 100;
            entries = {
              MenuEncryptObject
            }
          }
          TaskPaneSelectedExpansion {
            target = objchildren;
            condition = expression { cooobj.isencrypted }
            priority = 50;
            entries = {
              MenuDecryptObject
            }
          }
          MenuContextExpansion {
            condition = expression { !cooobj.isencrypted }
            entries = {
              MenuEncryptObject
            }
          }
          MenuContextExpansion {
            condition = expression { cooobj.isencrypted }
            entries = {
              MenuDecryptObject
            }
          }
        }
      }
    }
  }
}
```

When using the expansions which are targeting the container object (`TaskPaneExpansion`, `*DirectExpansion`) the current selection can be retrieved by using the current selection context.

The priority property of expansions is used to define the order of the expansions. Expansions with a higher priority value are displayed before those with a lower priority.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COODESK@1.1;
  extend class Order {
    menus {
      default = {
        expansions {
          ToolsMenuDirectExpansion {
            target = myattachments;
            condition = expression { count(#TV.TV_SELECTIONCONTEXT.selobjects) > 0 }
            entries = {
              MenuSendAllObjects
            }
          }
        }
      }
    }
  }
}
```

Expert note: Expansion points are just dynamic menus which are defined as group so that the calculated menu items are surrounded with separator lines.

7.4.4 Assigning task panes to an object class

Syntax

```
extend class reference {
  taskpanes {
    default = taskpane;
    scope = taskpane;
    ...
  }
}
```

Task panes are displayed in the left part of the so called quick view. By default, the quick view is launched when an activity in a user's work list is opened by a double-click. For an activity, the quick view shows the task pane defined for object class *Activity* as well as the task pane defined for the object class of the business object associated with the process.

Task panes are assigned to object classes using a `taskpanes` block that must be nested within an `extend class` block.

Each object class can have a default task pane which is defined using the `default` keyword.

You can also assign one or more scope-specific task panes to an object class. If there is no scope-specific task pane available in an object class for the current scope, the default task pane is displayed instead.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  extend class Order {
    taskpanes {
      default = TskPnOrder;
  }
}
```

To integrate the quick view for instances of an object class other than *Activity*, `COODESK@1.1:OpenObject` must be implemented with either the `FSCVENV@1.1001:StartEnterObjectApp` application or the `FSCVENV@1.1001:StartEnterObjectInNewWindowApp` application in this object class. The following example demonstrates how to add the quick view to the *Order* object class so it is opened when an order is double-clicked by a user.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COODESK@1.1;
  OpenObject {
    variant Order {
      impl = StartEnterObjectInNewWindowApp;
    }
  }
}
```

7.4.5 Assigning forms to an object class

Fabasoft Folio allows you to assign forms to an object class. These forms must be bound to specific use cases. When a use case involving a user interface is invoked, Fabasoft Folio tries to locate a matching form for this use case in the list of form bindings in the object class of the current object.

The most common use cases involving a user interface are listed in the next table. When creating a new object class, it is recommended to define form bindings for at least the first four use cases listed in the table.

Use case	Description
`COOSYSTEM@1.1:ObjectConstructor`	This use case is invoked when a new instance of an object class is created.
`COOATTREDIT@1.1:ReadObjectAttributes`	This use case is invoked when the properties of an object are read.
`COOATTREDIT@1.1:EditObjectAttributes`	This use case is invoked when the properties of an object are edited.
`COOSEARCH@1.1:SearchObjects`	This use case is invoked when the search dialog box is opened.
`COODESK@1.1:DisplayOptions`	The form assigned to this use case is used for allowing users to select columns when changing the column settings of an object list.
`COODESK@1.1:ExploreObject`	This use case is invoked when an object is opened in the explore view by selecting the *Explore* menu.
`COODESK@1.1:ExploreTree`	The form assigned to this use case is used when a compound object is displayed in the tree view.

Table 17: Use cases for form bindings

Note:

`COOSYSTEM@1.1:ObjectConstructor` replaces the mapping of `COOTC@1.1001:InitGUI`.

The following example illustrates different scenarios for object creation. The default implementations used in the example show the form that is mapped to `COOSYSTEM@1.1:ObjectConstructor`.

Example

```
// Application executed when an object is created via menu or
// button in an object list
override FSCVENV@1.1001:InitializeCreatedObject {
  variant objectclass {
    impl = FSCVENV@1.1001:InitializeCreatedObjectApp;
  }
}
// Application executed when an object is created inside of
// an object pointer property
override FSCVENV@1.1001:InitializeCreatedObjectDoDefault {
  variant objectclass {
    impl = FSCVENV@1.1001:InitializeCreatedObjectApp;
  }
}
// Application executed when an object is created from
// a template
override FSCVENV@1.1001:InitializeTemplateCreatedObject {
  variant objectclass {
    impl = FSCVENV@1.1001:InitializeCreatedObjectApp;
  }
}
```

The form bindings for an object class are defined in a `form` block that must be nested within the `extend class` block.

Syntax

```
extend class reference {
  forms {
    usecase {
      form;
      ...
    }
    ...
  }
}
```

The `form` block can contain one or more form bindings. Each binding consists of the reference of the use case, and a block denoting the forms that should be bound to this use case. In each binding, you can reference one or more forms.

The reason for referencing more than one form in a binding is that you might want to provide different forms for administrators and end-users. As a rule of thumb, remember that the administrator form must be listed before the end-user form in a form binding.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COODESK@1.1;
  import COOATTREDIT@1.1;
  import COOSEARCH@1.1;
  import COOTC@1.1001;
  extend class Order {
    forms {
```

```
      OpenObject {
        OrderAdminForm;
        OrderUserForm;
      }
      ReadObjectAttributes {
        OrderAdminForm;
        OrderUserForm;
      }
      EditObjectAttributes {
        OrderAdminForm;
        OrderUserForm;
      }
      SearchObjects {
        OrderSearchForm;
      }
      InitGUI {
        OrderCreateForm;
      }
    }
  }
}
```

Moreover, the use case `COODESK@1.1:OpenObject` is used for both the form binding for objects that are opened by a double-click and the explore view binding. In this case, the order of the forms in the `COODESK@1.1:OpenObject` block is not relevant.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCFOLIO@1.1001;
  import COODESK@1.1;
  import COOATTREDIT@1.1;
  import COOTC@1.1001;
  extend class ContactPerson {
    forms {
      OpenObject {
        // The following form is used when a contact person is double-
        // clicked by the user (i.e. the "Open" menu is invoked)
        FormCustomer;
        // The desk form is used when a contact person is opened in
        // "Explore" mode (e.g. it is selected in the tree view)
        DeskFormCustomer;
      }
      ReadObjectAttributes {
        FormCustomer;
      }
      EditObjectAttributes {
        FormCustomer {
          viewpriority = 100;
          viewcondition<condcondition> = {
            {
              expression { cooobj.validtill > coonow }
            }
          }
        }
      }
      SearchObjects {
        SearchFormCustomer;
      }
      InitGUI {
        CtorFormCustomer;
      }
    }
  }
}
```

The property `COOSYSTEM@1.1:viewpriority` is a priority value from 0 to n, used to rank the lines of the aggregate `COOSYSTEM@1.1:classviews`. The highest value determines the used view.

7.5 Defining portals

Syntax

```
portal reference {
  pane reference {
    colorscheme = colorscheme;
    page reference {
      application = application;
    }
  }
  ...
}
```

The app.ducx user interface language allows you to define custom portals in your software component.

In Fabasoft Folio, a portal is comprised of three distinct elements:

- the *Portal* (`FSCVPORT@1.1001:Portal`) object itself
- one or more *Portal Panes* (`FSCVPORT@1.1001:Pane`) that are part of the portal
- one or more *Portal Pages* (`FSCVPORT@1.1001:Page`) that are part of a portal pane

The keyword `portal` is used to define a portal. It must be followed by a reference and curly braces. The portal may contain multiple portal panes, which are defined inside the `portal` block using the `pane` keyword.

Each `pane` block must have a reference followed by curly braces. Inside the `pane` block, you may use the `colorscheme` keyword to assign a predefined color scheme to the portal pane.

The `page` keyword denotes a portal page. Multiple pages can be nested within a `pane` block. In each `page` block, a virtual application must be referenced using the `application` keyword. This virtual application is invoked when the portal page is displayed.

Example

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCVAPP@1.1001;
  portal CustomerPortal {
    pane OrderPane {
      colorscheme = Green;
      page OrderPage {
        application = OrderOverview;
      }
    }
  }
}
```

8 app.ducx Use Case Language

The purpose of the app.ducx use case language is to define and implement use cases. With the app.ducx use case language you can define new use cases and provide method implementations

for these use cases. You can use app.ducx expression language or virtual applications to implement use cases. Alternatively you can use Java for implementing.

A use case model block consists of import declarations, transaction variable declarations, and use case model elements. The `usecases` keyword denotes a use case model block. It must be followed by the reference of your software component and curly braces.

Use case model blocks can only be contained in files with a `.ducx-uc` extension.

Syntax

```
usecases softwarecomponent
{
  // Import declarations
  import softwarecomponent;
  // Transaction variable declarations
  variables {
    datatype reference;
    ...
  }
  // Use case model elements
  ...
}
```

8.1 Declaring transaction variables

Syntax

```
variables {
  datatype reference;
  ...
}
```

In the `usecases` block, you can use `variables` keyword to declare a list of transaction variables provided by your software component. It must be followed by curly braces, and a list of transaction variable declarations.

Note: The `variables` block can be omitted, if your software component does not provide its own set of transaction variables. However, if present the `variables` block must be the first block within the `usecases` block.

Each transaction variable declaration consists of a valid Fabasoft app.ducx data type, and the reference of the transaction variable to be declared.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  variables {
    boolean TV_PRINTINVOICE;
    Invoice TV_INVOICE;
  }
}
```

8.2 Defining use cases

Fabasoft Folio supports different types of actions that can be invoked on an object:

- An *Action* (`COOSYSTEM@1.1:Action`) is considered to be the declaration of a private method that is implemented on one or more object classes. Actions can be marked as public.

- A *Use Case* (`COOSYSTEM@1.1:UseCase`) is considered to be the declaration of a public method. It can provide different implementations on one or more object classes. Use cases can be marked as private.

For more information on public, private and obsolete see chapter 4.2.10 “Public, private and obsolete”.

This section describes the definition of use cases using the app.ducx use case language of Fabasoft app.ducx.

8.2.1 Defining a new use case

Syntax

```
usecase reference(parameter, ...) {
  accexec = accesstype;
  // Providing the implementation for one object class
  variant objectclass {
    impl = ...
  }
  // Providing the same implementation for more object classes
  variant objectclass1, objectclass2 {
    impl = ...
  }
}
```

The `usecase` keyword is used to declare a use case. It must be followed by the reference and by parentheses, holding the list of parameters or the prototype.

Note: An action is declared the same way but omitting the `usecase` keyword.

8.2.1.1 Defining the list of parameters

Parameters must be separated by commas. For each parameter, the data type must be specified. For objects, use the `object` keyword when denoting the data type. For specifying an arbitrary data type, use the `any` keyword. Square brackets can be used to denote lists.

By default, parameters are treated as input parameters. Output parameters must be prefixed with the keyword `out`. For input/output parameters, the prefix `ref` must be used. The keywords `retval out` (`out` may be omitted) and `retval ref` denote a parameter that is used as the default return value, if there are multiple output parameters. The keyword `optional` can be used to denote an optional parameter. `unique` denotes a unique list in a parameter.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  usecase CalcOrderPosValue(integer quantity, currency unitprice,
    out currency total, optional float rebate) {
    ...
  }
  usecase CalcOrderTotal(OrderPosition[] positions, boolean includetax,
    out currency total, out currency tax) {
    ...
  }
  usecase ValidateOrderPositions(ref unique OrderPosition[] positions) {
    ...
  }
  usecase CreateCollectiveInvoice(Order[] orders, retval Invoice invoice) {
    ...
  }
```

```
}
```

8.2.1.2 Assigning a prototype instead of a parameter list

Prototypes are component objects containing predefined sets of parameters. Certain use cases – such as property triggers described in chapter 5.3.3 “Assigning triggers to a property” – require predefined prototypes in order to function correctly.

For assigning a prototype to a use case instead of specifying a list of parameters, you can use the `parameters as` keywords followed by the reference of the prototype.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  // Skeleton of a get value trigger action
  GetOrderTotal(parameters as AttrGetPrototype) {
    ...
  }
  // Skeleton of a set value trigger action
  SetOrderPositions(parameters as AttrSetPrototype) {
    ...
  }
}
```

8.2.1.3 Protecting a use case with an access type

The `accexec` keyword is used to assign an access type to a use case.

If protected by an access type, a use case can only be invoked if the user trying to invoke it is granted the access type by the object’s ACL. However, it is not mandatory to protect a use case with an access type. Please refer to chapter 9.4 “Defining an access type” for further information on how to define access types.

8.2.1.4 Providing the implementation for a use case

In order to be able to invoke a use case, you have to provide an implementation for one or more object classes. You may also provide different implementations of the same use case for different object classes just as in an object-oriented programming language, Fabasoft app.ducx allows an object class to provide a more specific implementation of a use case that is already implemented by one of its base classes. In this case, the implementation in the subclass overrides the implementation in the base class.

The Fabasoft Folio Kernel returns an error message if a use case is invoked on an object where no implementation is found in the object class of the object or in one of its base classes.

The `variant` keyword is used to denote the object class for which an implementation of a use case is provided. In a `variant` block, the `impl` keyword is used to assign an implementation of the use case to the object class. It may be followed by the `application`, `java`, `expression` or `empty` keyword:

- The `application` keyword specifies a virtual application that is called when the use case is invoked. A virtual application is required when your use case implementation must display user interface elements. Please refer to chapter 8.3 “Defining a virtual application” for further information on implementing virtual applications.
- The `java` keyword denotes a Java use case implementation. Please refer to chapter 8.4 “Implementing a use case in Java” for further information on implementing Java use cases.
- The `expression` keyword defines a use case implementation in app.ducx expression language.

- The `empty` keyword defines an empty implementation that does nothing.

If you want to provide the same implementation for multiple object classes, you can enumerate these object classes in a comma separated list after the `variant` keyword.

In a `variant` block, you may also define optional use case hints to describe the behavior of the use case implementation. To define use case hints, the `hints` keyword must be denoted followed by curly braces and a list of use case hints as described by the next table.

Hint	Description
`MH_CHANGESOBJ`	The use case implementation changes the current object.
`MH_NEEDSSERVER`	The use case implementation requires a connection to the server.
`MH_NEEDSARCHIVE`	The use case implementation requires an archive implementation.
`MH_NEEDSLOCALGUI`	The use case implementation requires interaction with the graphical user interface.
`MH_CHANGESVIEW`	The use case implementation changes the current view.
`MH_CREATESOBJ`	The use case implementation creates a new object in the currently selected object pointer or object list.
`MH_NEEDSEXPLORER`	The use case implementation requires explore mode.

Table 18: Use case hints

8.2.1.5 Implementing a use case in app.ducx expression language

The app.ducx expression language can be used for implementing a use case. When implementing a use case as expression, the current object – that is the object on which the use case has been invoked – can be accessed by the `cooobj` variable. Additionally, the `coort` interface object can be used for accessing the Fabasoft Folio Runtime, and the `cootx` variable contains an interface to the current transaction. Parameter names can be used to access the parameters declared in the use case signature.

Note: The app.ducx compiler verifies the syntax of expressions, but it cannot verify semantic correctness.

8.2.1.5.1 Implementing a use case as an inline expression

In the following example, the use case `CreateInvoice` is implemented in object class `APPDUCXSAMPLE@200.200:Order` as an expression. The expression creates a new instance of object class `APPDUCXSAMPLE@200.200:Invoice`, initializes some of its properties, passes back the invoice object in parameter `invoice`, and references the new invoice in the `APPDUCXSAMPLE@200.200:invoice` property of the order.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  usecase CreateInvoice(out Invoice invoice) {
    variant Order {
      impl = expression {
        Invoice invoice = #Invoice.ObjectCreate()[2];
        invoice.invoiceorder = cooobj;
```

```
        invoice.invoicedate = coonow;
        invoice.COODESK@1.1:ShareObject(null, null,
            #APPDUCXSAMPLE@200.200:invoice, cooobj);
      }
      hints = {
        MH_NEEDSSERVER,
        MH_CHANGESOBJ
      }
    }
  }
}
```

If a use case overrides one of its superclass's use cases, you can invoke the overridden use case through the use of the keyword `super`. `super` can only be called when `coometh` (holds the so-called method context) is available. `in` parameters that are changed before the call of `super` are passed in the changed state. `out` parameters are written to the local scope after the call of `super`.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  usecase CallSuper(integer inx, retval string name) {
    variant Object {
      impl = expression {
        name = inx;
      }
    }
    variant CompoundObject {
      impl = expression {
        inx++;
        super();
        %%TRACE("name = ", name);
      }
    }
  }
}
```

8.2.1.5.2 Implementing a use case in a separate app.ducx expression language file

Fabasoft app.ducx allows you to store the actual expression code in a separate app.ducx expression language file.

Example

app.ducx Use Case Language

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  usecase CreateInvoice(out Invoice invoice) {
    variant Order {
      impl = file("CreateInvoice.ducx-xp");
    }
  }
}
```

app.ducx Expression Language (CreateInvoice.ducx-xp)

```
#APPDUCXSAMPLE@200.200:Invoice.ObjectCreate(null, &invoice);
invoice.APPDUCXSAMPLE@200.200:invoiceorder = cooobj;
invoice.APPDUCXSAMPLE@200.200:invoicedate = coonow;
```

8.2.2 Defining a new menu use case

Syntax

```
menu usecase reference on selected {
  variant objectclass {
    impl = ...
  }
}
menu usecase reference on selected or container {
  variant objectclass {
    impl = ...
  }
}
menu usecase reference direct {
  variant objectclass {
    impl = ...
  }
}
```

A menu use case is a special type of use case that is usually invoked by a user selecting a menu item in the user interface.

There are three options available, specified by using keywords after the reference:

- `on selected`: The menu use case is invoked on each object selected by the user.
- `on selected or container`: The menu use case is invoked on each object selected by the user. However, if no objects have been selected, the menu use case is invoked on the container object instead. For example, if you are invoking a menu use case from within a folder but no objects have been selected in the folder, it is invoked on the folder itself.
- `direct`: The menu use case is invoked on the container object only. It is NOT invoked on any objects selected by the user. However, it is possible to get the currently selected objects by using `COODESK@1.1:GetSelected` from within the implementation of the menu use case. If the use case is implemented as virtual application, the selected objects are also available in the `sys_selobjects` parameter.

To enable this functionality, a number of elements are required. These are generated automatically by the app.ducx compiler:

- the menu object that can be added to a menu or a context menu,
- the outer use case that is invoked on the container object (when using the `on selected` or `on selected or container` clause),
- the implementation of the outer use case using method definition `COODESK@1.1:CallSelectedObjectsMethod` in object class `COOSYSTEM@1.1:Object` (when using the `on selected` or `on selected or container` clause),
- the application mapping of the outer use case to `FSCVENV@1.1001:DoSelected` (when using the `on selected` or `on selected or container` clause), and
- the inner use case that is invoked on each of the selected objects.

The following example shows the definition of a menu use case for setting the state of an instance of object class `APPDUCXSAMPLE@200.200:Order` to `OS_SHIPPED`. This menu use case is invoked on each of the order objects selected by the user. For this menu use case, the Fabasoft app.ducx compiler also generates a menu object with the reference `APPDUCXSAMPLE@200.200:MenuMarkOrderAsShipped`. In the example, this menu object is added to context menu `APPDUCXSAMPLE@200.200:MenuRootOrderContext` to allow users to invoke the use case from the context menu of an order to change the state of the order to `OS_SHIPPED`.

Note: Objects must be locked before they may be changed. To lock an object, `COOSYSTEM@1.1:ObjectLock` must be invoked on the object.

Example

app.ducx Use Case Language

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  menu usecase MarkOrderAsShipped on selected {
    // Use defined symbol in ressources file
    symbol = SymbolMarkOrderAsShipped;
    variant Order {
      impl = expression {
        cooobj.ObjectLock(true, true);
        cooobj.orderstate = OrderState(OS_SHIPPED);
      }
    }
  }
}
```

app.ducx User Interface Language

```
userinterface APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COODESK@1.1;
  // Context menu for object class APPDUCXSAMPLE@200.200:Order
  menuroot MenuRootOrderContext {
    entries = {
      MenuEditCopy,
      MenuSeparator,
      MenuObjectUnshare,
      MenuSeparator,
      MenuOpenObjectAttributes,
      MenuSeparator,
      MenuMarkOrderAsShipped // Add menu for APPDUCXSAMPLE@200.200:MarkOrderAsShipped
    }
  }
  // Assign context menu APPDUCXSAMPLE@200.200:MenuRootOrderContext to object class
  // APPDUCXSAMPLE@200.200:Order
  extend class Order {
    menus {
      default = {
        ctxmenu = MenuRootOrderContext;
      }
    }
  }
}
```

Note: Menu use cases can also be implemented in Java or as a virtual application. If implemented as virtual application, a predefined prototype is assigned to the virtual application:

For menu use cases employing the `on selected` or `on selected or container` clause the `FSCVENV@1.1001:MenuPrototype` prototype is assigned to the virtual application, and the parameters described in the next table are passed to it.

Variable	Description
`venv_object`	The variable `venv_object` stores the object the use case was invoked on.
`venv_parent`	The variable `venv_parent` stores the container from which the use case was executed.

venv_index	The variable venv_index stores the index of the object in the property from which the use case was executed.
venv_view	The variable venv_view stores the property from which the use case was executed.
venv_action	The variable venv_action stores the use case that is executed.

Table 19: Parameters of FSCVENV@1.1001:MenuPrototype

For menu use cases employing the direct clause the FSCVAPP@1.1001:MenuPrototype prototype is assigned to the virtual application, and the parameters described in the next table are passed to it.

Variable	Description
sys_object	The variable sys_object stores the object the use case was invoked on.
sys_action	The variable sys_action stores the use case that is executed.
sys_view	The variable sys_view stores the property from which the use case was executed.
sys_selobjects	The variable sys_selobjects stores the objects selected in the property stored in variable sys_view.
sys_selindices	The variable sys_selindices stores the index of each selected object in the property stored in variable sys_view.
sys_dynkey	The variable sys_dynkey stores the key of the dynamic menu and is only available when the use case was executed from a dynamic menu.

Table 20: Parameters of FSCVAPP@1.1001:MenuPrototype

8.3 Defining a virtual application

Use cases and menu use cases can be implemented as virtual applications.

If at any point your use case needs to present a user interface – such as a form or a dialog box – you are required to implement it as a virtual application. This is also the case when you invoke another use case requiring user interaction from your use case.

A virtual application is composed of expressions and dialogs. Dialogs allow you to display a form or form page, and have a set of branches.

When a use case is implemented as virtual application, the Fabasoft app.ducx compiler generates an instance of object class *Application* (FSCVAPP@1.1001:Application) for the virtual application, usually with the same reference as specified for the use case. Moreover, an instance of object class *Application View* (FSCVAPP@1.1001:ApplicationView) is generated for each dialog defined.

The application keyword denotes the definition of a virtual application. An application block can contain declarations of global variables, an expression block, multiple inline definitions of dialogs and generic assignments. Each dialog must be defined within its own dialog block embedded in the application block. In order to be displayed, a dialog must be called explicitly from within the expression block using the detachment operator ->.

Global variables must be declared before the expression block by denoting data type and the variable name followed by a semicolon. Once declared, global variables are available within the scope of the entire virtual application including all its dialogs. For example, if you need to increase a

counter as your virtual application is processed you should to declare a global variable for the counter.

In addition to global variables, you can also reference global fields that are available throughout your virtual application. Global field must be declared before the expression block. The keyword `field` must be used for declaring a global field. It must be followed by its reference and a semicolon.

Note: You may only reference fields that have been defined in a `fields` block using the Fabasoft app.ducx object model language.

Syntax

```
usecase reference(parameter, ...) {
  variant objectclass {
    impl = application {
      // Virtual applications can consist of global variable declarations,
      // global fields, expression blocks and dialogs that can be defined
      // inline
      datatype variable;
      field reference;
      ...
      expression {
        ...
      }
      overlaysize = overlaysizevalue;
      dialog reference dialogmodifiers {
        // Initialization expression that is executed before the dialog
        // is displayed
        expression {
          ...
        }
        form = form;
        target = target;
        value = valuetype;
        autoload = booleanvalue;
        autostore = booleanvalue;
        autolock = booleanvalue;
        catcherrors = booleanvalue;
        overlaysize = overlaysizevalue;
        branch identifier branchmodifiers {
          caption = captionstring;
          description = descriptionstring;
          symbol = symbol;
          visible = expression {
            ...
          }
          expression {
            ...
          }
          applytofield<view, identopt> = {
           {form, field}
          }
        }
        ...
        behaviors {
          behavior {
            condition = expression {
              ...
            }
            applytofield<view, identopt> = {
              {form, field},
              ...
            }
          }
          ...
        }
      }
    }
  }
```

```
}
```

When a use case that is implemented as virtual application is called, the virtual application is invoked by the Fabasoft Folio vApp Engine, and the expressions defined within the `expression` block are executed by the Fabasoft Folio Kernel.

8.3.1 Implementing a virtual application

A virtual application is implemented as a list of statements written in app.ducx expression language.

As usual, the `cooobj` variable holds the current object on which the use case is invoked. `cootx` and `coort` are also available for accessing the current transaction and the Fabasoft Folio Runtime.

8.3.1.1 Accessing parameters from a virtual application

The parameters expected by the virtual application must be specified in the parameter list of the use case, and are instantly available to you as variables in the expression code.

In the example, a use case for adding a new entry to the list of order positions of an order is implemented as a virtual application. The parameters `product` and `quantity` – that must be supplied when the use case is invoked – are instantly available as variables in the `expression` block. Moreover, the detachment operator, which is discussed later in this chapter, is used to invoke an existing use case for editing the properties of the order.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  usecase AddPositionToOrder(Product product, integer quantity) {
    variant Order {
      impl = application {
        expression {
          cooobj.ObjectLock(true, true);
          OrderPosition position = OrderPosition();
          position.product = product;
          position.quantity = quantity;
          cooobj.orderpositions += position;
          cooobj->COOATTREDIT@1.1:EditObjectAttributes();
        }
      }
    }
  }
}
```

8.3.1.2 Invoking use cases, virtual applications and dialogs

Expressions that are hosted within a virtual application or within a dialog can make use of the detachment operator `->` in order to invoke another use case, a virtual application, or a dialog.

Note: Instead of statically specifying the use case, virtual application or dialog, you can use square brackets for dynamic evaluation of the use case, virtual application or dialog to be invoked.

Example

```
// static invocation of a use case
cooobj->COODESK@1.1:OpenObject();
// dynamic invocation of a use case
UseCase uc = #COODESK@1.1:OpenObject;
cooobj->[uc]();
```

8.3.1.2.1 Invoking a dialog

For invoking a dialog, the detachment operator `->` must be followed by the reference of the dialog to be displayed. A dialog does not have any parameters. Supplying parameters when invoking a dialog leads to a runtime error.

Using the detachment operator, you can invoke any of the dialogs defined in the `application` block of your virtual application. You can also invoke dialogs that have been defined in other applications for reasons of reusability. However, reusing dialogs is strongly discouraged since unlike use cases and virtual applications they are not self-contained units with a defined interface.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  menu usecase CreateOrderWizard on selected {
    variant ContactPerson {
      impl = application {
        integer currentstep; // Global variable for storing step index
        Order neworder;      // Global variable for storing the new order
        expression {
          currentstep = 1;
          ->WizardStep1;
        }
        dialog WizardStep1 {
          ...
        }
        dialog WizardStep2 {
          ...
        }
        dialog WizardStep3 {
          ...
        }
      }
    }
  }
}
```

8.3.1.2.2 Invoking a virtual application

A virtual application can be invoked with the detachment operator `->` followed by the reference of the virtual application and the list of parameters enclosed in parentheses. Parameters may not be omitted. Thus, all parameters defined in the parameter list of the virtual application to be invoked must be specified. However, you can use `null` if you do not want to provide a value for an optional parameter.

Note: You should favor invoking a use case instead of a virtual application, whenever possible (see chapter 8.3.1.2.3 "Invoking another use case").

The following example demonstrates invoking the virtual application `FSCVENV@1.1001:EditObjectAttributeApp`.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  menu usecase EditInvoice on selected {
    variant Order {
      impl = application {
        expression {
          Invoice invoice = cooobj.orderinvoice;
          if (invoice != null) {
            ->FSCVENV@1.1001:EditObjectAttributeApp(invoice, cooobj, null,
```

```
              #orderinvoice, null, false, null, false);
          }
          else {
            throw coort.SetError(#NoInvoiceFound, null);
          }
        }
      }
    }
  }
}
```

8.3.1.2.3 Invoking another use case

When invoking another use case from a virtual application, you have to specify an object on which the use case is to be called. The object must be followed by the detachment operator, the reference of the use case to be called, and the list of parameters enclosed in parentheses.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  usecase OpenInvoice() {
    variant Order {
      impl = application {
        expression {
          Invoice invoice = cooobj.orderinvoice;
          if (invoice != null) {
            invoice->COODESK@1.1:OpenObject();
          }
          else {
            throw coort.SetError(#NoInvoiceFound, null);
          }
        }
      }
    }
  }
}
```

8.3.1.3 Special variables in virtual applications

When a virtual application or a dialog is invoked, the Fabasoft vApp engine – depending on the context – makes available a set of special variables in the local scope `this`. As already stated, some of these variables will only be available in certain situations, e.g. when the user executes a branch. These special variables are described in the next table.

Variable	Description
`sys_object`	The variable `sys_object` stores the current object or the container object when a menu is invoked.
`sys_action`	The variable `sys_action` stores the use case or action that is executed.
`sys_view`	The variable `sys_view` stores the currently selected property when a menu or a branch is executed.
`sys_selobjects`	The variable `sys_selobjects` stores the list of objects selected in the list associated with the executed branch.
`sys_selindices`	The variable `sys_selindices` stores the list of indices selected in the list

	associated with the executed branch.
`sys_branchvalue`	The variable `sys_branchvalue` stores the current value of the data field associated with the executed branch.
`sys_branchattr`	The variable `sys_branchattr` stores the full path to the property of the data field associated with the executed branch.

Table 21: Special variables in virtual applications

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCFOLIO@1.1001;
  menu usecase ShipSelectedOrders on selected {
    variant ContactPerson {
      impl = application {
        expression {
          ->SelectOrder;
        }
        dialog SelectOrder {
          form = SelectOrders;
          target = cooobj;
          cancelbranch;
          nextbranch {
            expression {
              if (sys_selobjects > 0) {
                sys_selobjects-> ShipOrder();
              }
              else {
                throw coort.SetError(#NoObjectsSelected, null);
              }
            }
          }
        }
      }
    }
  }
}
```

8.3.1.4 Virtual application context actions

The action `FSCVAPP@1.1001:GetVAPPInformation` returns a dictionary in the first parameter, which contains the keys listed in the next table.

Key	Description
`application`	The `application` key stores the virtual application that is currently running.
`dispatcher`	The `dispatcher` key stores the application dispatcher used.
`servertype`	The `servertype` key allows you to determine the type of web server used. For example, the value “IIS” is stored for Microsoft Internet Information Services.
`iswai`	The `iswai` key allows you to determine whether the virtual application is accessibility-enabled.
`isajax`	The `isajax` key allows you to determine whether AJAX is enabled on the client side.

iswindows	The iswindows key allows you to determine whether the Fabasoft Folio Web Service is hosted on a Microsoft Windows computer.

Table 22: Virtual application context dictionary

The action FSCVAPP@1.1001:GetContext allows you to access the virtual application context interface exposing the GetServerVariable and GetFormVariable methods.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  usecase GetvAppInfos() {
    variant Object {
      impl = application {
        expression {
          vappctx = coouser.FSCVAPP@1.1001:GetContext()[1];
          if (vappctx != null) {
            cookies = vappctx.GetServerVariable("HTTP_COOKIE");
            querystring = vappctx.GetServerVariable("QUERY_STRING");
            useragent = vappctx.GetServerVariable("HTTP USER AGENT");
            remoteuser = vappctx.GetServerVariable("REMOTE USER");
            remoteaddress = vappctx.GetServerVariable("REMOTE ADDR");
            formvar_firstname = vappctx.GetFormVariable("firstname");
            formvar_surname = vappctx.GetFormVariable("surname");
            %%TRACE("Local variables", this);
          }
        }
      }
    }
  }
}
```

The following example demonstrates how to post an HTML form to a virtual application (assuming a default installation of the Fabasoft Folio Web Services on a host named “folioweb”).

In the ax parameter, the address of the virtual application must be provided (for the example, assume that COO.200.200.1.1000 is the address of the GetvAppInfos virtual application from the previous example). The return URL that is invoked after submitting the HTML form must be provided in the ru parameter.

Note: For the HTML field values to become available in the virtual application context, the URL parameter xhr=s must be added to the virtual application URL.

Example

```
<html>
  <body>
    <form action="http://folioweb/fsc/fscasp/content/bin/fscvext.dll?
      ax=COO.200.200.1.1000&ru=http%3A%2F%2Fwww.fabasoft.com&xhr=s" method="post">
      <label for="firstname">First Name: </label>
      <input type="text" name="firstname"><br>
      <label for="surname">Surname: </label>
      <input type="text" name="surname"><br>
      <input type="submit" value="Send"> <input type="reset">
    </form>
  </body>
</html>
```

8.3.1.5 Direct assignments to a virtual application

8.3.1.5.1 Defining overlay size

The overlay size for a application can be defined by using a generic assignment of the enumeration type `FSCVAPP@1.1001:OverlaySize` to the property `FSCVAPP@1.1001:overlaysize`. This setting is used for all dialogs in the virtual application except a dialog has an own definition.

8.3.2 Extending a virtual application

A virtual application can be extended with dialogs in context of a use case and variant. Use the `using` keyword followed by the reference of the generated virtual application (for the combination of variant and use case) and curly braces.

Syntax

```
using reference {
  dialog reference {
    ...
  }
}
```

In the following example the virtual application for the menu use case “EditInvoice” that is defined for variant “Order” gets extended by the “EditOrder” dialog.

Example

```
using OrderEditInvoice {
  dialog EditOrder {
    form = expression {
      if (coort.GetCurrentUserRoleGroup() == #SysAdm) {
        return #FormEditOrderAdmin;
      }
      else {
        return #FormEditOrderUser;
      }
    }
  }
}
```

8.3.3 Defining a dialog

A dialog is defined within a `dialog` block nested within the `application` block of your use case. You can define multiple dialogs within an `application` block.

Note: The `dialog` blocks must not precede the `expression` block within an `application` block. The `expression` block must be the first block within an `application` block.

For a comprehensive example illustrating the definition of a dialog, please refer to the end of this section.

8.3.3.1 Assigning a form

You can use the `form` keyword to assign a form or form page to a dialog, which is shown when the dialog is invoked.

Note: Refer an instance of object class *View* (`FSCVIEW@1.1001:View`) or *Tabbed View* (`FSCVIEW@1.1001:TabbedView`) is not supported any more.

The form to be assigned to a dialog can also be dynamically calculated using app.ducx expression language. When an app.ducx expression is used, it must return the form, form page or view to be displayed on the dialog.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  menu usecase EditOrder on selected {
    variant Order {
      impl = application {
        expression {
          ->EditOrder
        }
        dialog EditOrder {
          form = expression {
            if (coort.GetCurrentUserRoleGroup() == #SysAdm) {
              return #FormEditOrderAdmin;
            }
            else {
              return #FormEditOrderUser;
            }
          }
          overlaysize = OVERLAYSIZE_LARGE;
        }
      }
    }
  }
}
```

8.3.3.2 Choosing the view mode

A dialog has two view modes: edit mode and read-only mode. When in edit mode, users are allowed to enter values into the fields displayed on the form. Otherwise, fields are not editable in the user interface.

By default, each dialog defined is displayed in edit mode. If you want to define a read-only dialog, the `readonly` dialog modifier suffix must be denoted following the dialog's reference.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  menu usecase ViewInvoice on selected {
    variant Invoice {
      impl = application {
        expression {
          ->ViewInvoice
        }
        dialog ViewInvoice readonly {
          ...
        }
      }
    }
  }
}
```

8.3.3.3 Enabling query mode

An dialog has two view types: value mode and query mode. Query mode is a special mode for creating query forms. To enable query mode, the `value` keyword must be set to `VALUETYPE_QUERY`.

When in query mode, a dialog must be initialized with a query following Fabasoft F-SQL syntax. This query must be passed to the dialog in the `sys_query` system variable. The user can then use the dialog to enter additional query conditions, and the resulting query is returned in the `sys_query` system variable.

8.3.3.4 Assigning a target object

For each dialog, you can assign a target object on which it is operating. The Fabasoft Folio vApp Engine provides a mechanism for automatically loading and storing property values from and to the target object. The `target` keyword is used to assign a target object to a dialog. Usually, the name of a variable holding the desired target object is specified as the target. Additionally the target can be defined as app.ducx expression.

8.3.3.5 Automatic loading and storing of field values

By default, property values are read from the database and displayed in the corresponding fields on the form when a dialog is invoked.

When the user leaves the dialog by executing a branch, changes by the user are automatically saved – provided that the branch expression discussed later does not throw an exception.

This feature, however, requires that a target object is assigned to a dialog using the `target` keyword.

Fabasoft app.ducx allows you to suppress automatic loading of field values by setting `autoload` to `false` for a dialog. Conversely, the `autostore` keyword can be used to disable storing field values when leaving a dialog.

For instance, `autostore` should be set to `false` for a dialog that is set to read-only view mode as it would not make sense to write back values to the target object that cannot be changed in the user interface anyway.

8.3.3.6 Defining overlay size

The overlay size for a dialog can be defined by using a generic assignment of the enumeration type `FSCVAPP@1.1001:OverlaySize` to the property `FSCVAPP@1.1001:overlaysize`.

8.3.3.7 Defining dialog behaviors

Behaviors allow you to customize the look and feel of a dialog or the fields on the form page displayed by the dialog.

Behavior definitions must be contained in a `behaviors` block nested within a `dialog` block. The `behaviors` keyword must be followed by curly braces.

Within the `behaviors` block, the keyword representing the corresponding behavior must be denoted, followed by curly braces. The same definition block can be used for multiple behaviors. In this case, the behavior keywords must be separated by commas. The next table shows a list of supported behaviors along with the respective keywords that must be used.

Additionally to the listed keywords all behaviors defined in the enumeration type *Hint* (`FSCVAPP@1.1001:HintType`) can be used. Instead of the keyword the *Reference Name* (e.g. `VIEWHINT_FLAT`) denotes the behavior.

Behavior	Description
`genericform`	The `genericform` behavior is used to specify a pattern to match an action that selects a form; the pattern may start with "explore:" to match an

	explore form. When applied to a field, `genericform` determines whether the generic built-in functionality for displaying the field – such as tooltips and context menus – should be enabled.
`symbol`	The `symbol` behavior is used to specify whether a row symbol should be displayed for the entries of lists.
`selection`	The `selection` behavior is used to specify whether the selection of one or more rows should be enabled for lists.
`required`	The `required` behavior is used to specify whether a field requires the user to enter a value.
`editcondition`	The `editcondition` behavior is used to specify a condition that must evaluate to `true` in order to make a field editable. If the condition returns `false`, the selected field is displayed in read-only mode.
`viewcondition`	The `viewcondition` behavior is used to specify a condition that must evaluate to `true` in order to make a field visible. If the condition returns `false`, the selected field is not displayed on the user interface.
`reference`	The `reference` behavior is used to specify that component object references should be displayed instead of the object names in the object name column of object lists.
`exploremode`	The `exploremode` behavior is used to specify that object lists should be displayed on separate pages on the user interface.
`simple`	The `simple` behavior is used to enable the simple view.
`showmultifunctionbar`	The `showmultifunctionbar` behavior is used to enable or disable the multifunctional bar. By default, the multifunctional bar is enabled.
`menus`	The `menus` behavior is used to enable or disable menu bars for a dialog. By default, menus are enabled.
`sort`	The `sort` behavior is used to enable or disable sorting for object lists and compound lists. By default, sorting is enabled.
`contextmenus`	The `contextmenus` behavior is used to enable or disable context menus for a dialog. By default, context menus are enabled.
`create`	The `create` behavior is used to enable or disable creating new instances for object pointer properties and object lists.
`search`	The `search` behavior is used to enable or disable search functionality for object pointer properties and object lists.

Table 23: Behaviors

Each behavior can be applied to one or more fields of a form page or the dialog itself. An `applytofield` initialization block must be used to apply a behavior block to a specific field or to a list of fields. `applytofield` is a compound property, but in many cases only the `identopt` property of the compound property is used. Therefore the value for `identopt` can be assigned directly to `applytofield` (e.g. `applytofield = "objname";`).

Furthermore, the `condition` keyword can be used to define a condition in app.ducx expression language. When a `condition` block is defined within a behavior block, the behavior is only applied if the expression returns `true`.

8.3.3.8 Defining branches

When described in an oversimplified manner, a branch just defines a button and a handler that is invoked when the button is clicked. In Fabasoft terminology, “clicking the button” is referred to as “executing the branch”.

The `branch` keyword must be used for defining a branch, followed by a unique branch identifier (which is also referred to as the programming name) and curly braces. Multiple `branch` blocks can be nested within a dialog.

8.3.3.8.1 Assigning branch modifiers

The branch identifier can be followed by optional branch modifiers. The next table shows a list of supported branch modifiers. If multiple branch modifiers are assigned to a branch, they must be separated by whitespaces.

Modifier	Description
`leave`	If the `leave` branch modifier is provided, the Fabasoft Folio vApp Engine continues processing the virtual application with the statement following the statement invoking the current dialog. If the `leave` branch modifier is omitted, the user is taken back to the current dialog after the branch has been executed successfully.
`default`	The `default` branch modifier should be applied to the default branch. There can only be one default branch for each dialog.
`hyperlink`	The `hyperlink` branch modifier allows you to attach a branch to a value displayed by a field on the form. The field’s value will then behave like a hyperlink, so a user can click on the value to activate the branch. A hyperlink branch does not have a caption.
`skipnull`	The `skipnull` branch modifier allows a user to invoke a branch even though no value was provided for one or more required fields on the form. When the `skipnull` branch modifier is omitted, a user is not able to invoke a branch before values are provided for all required fields.

Table 24: Branch modifiers

8.3.3.8.2 Assigning symbol, caption and description text

A symbol can be assigned to a branch using the `symbol` keyword. Please refer to chapter 6.3 “Defining symbols” for detailed information on how to define a new symbol.

The caption text displayed on a branch is usually loaded from a *String* (`COOSYSTEM@1.1:String`) object, which can be assigned to a branch using the `caption` keyword. Likewise, the `description` keyword is used to assign a description string to a branch, which is displayed when the user hovers the mouse over the branch. Please consult chapter 6.1 “Defining strings” for information on how to define new strings.

8.3.3.8.3 Ignoring field values

By default, the Fabasoft Folio vApp Engine does not execute a branch handler expression if not all required fields on the current form actually contain a value. Instead, an error message is displayed requesting the user to supply values for all required fields.

In order to force the Fabasoft Folio vApp Engine not to check whether required fields contain a value, you can use the `skipnull` branch modifier.

You will typically use `skipnull` in *Cancel* branches for aborting the execution of a virtual application.

8.3.3.8.4 Dynamically hiding a branch

The `visible` keyword allows you to define an app.ducx expression for evaluating whether a branch should be displayed. If the expression returns `true`, the branch is displayed, otherwise it is hidden.

8.3.3.8.5 Attaching a branch to a field

A branch can be attached to a field displayed on the form. An `applytofield` block must be used for attaching a branch to a field. `applytofield` is a compound property, but in many cases only the `identopt` property of the compound property is used. Therefore the value for `identopt` can be assigned directly to `applytofield` (e.g. `applytofield = "objname";`).

If a branch is attached to an object list or a compound list, it is displayed above the field. If it is attached to property containing a scalar value, it is displayed on the left-hand side next to the field.

Branches can also be attached to properties that are part of a compound property displayed on the form.

The `hyperlink` branch modifier allows you to attach the branch to the value displayed by a field. If it is omitted, the branch will be displayed next to the field value.

8.3.3.8.6 Defining the branch handler expression

You can use app.ducx expression language to implement a branch handler. The branch handler expression is defined in an `expression` block nested within the `branch` block.

You can use the detachment operator for invoking another dialog, or for executing another virtual application or a use case. For further information on how to use the detachment operator, please refer to chapter 8.3.1.2 “Invoking use cases, virtual applications and dialogs”.

8.3.3.8.7 Predefined branches

The majority of all dialogs contain two common branches: a “Next” branch for proceeding to the next page, and a “Cancel” branch for aborting the execution of the current virtual application. Therefore, Fabasoft app.ducx provides special keywords for these two common cases in form of predefined branches:

- The keyword `nextbranch` allows you define a “Next” branch for proceeding to the next dialog and continuing the execution of a virtual application. A `nextbranch` is implicitly assigned caption `FSCVENV@1.1001:StrNext` and symbol `COODESK@1.1:SymbolNext`. Moreover, the `leave` branch modifier is also implicitly assigned to a `nextbranch`.
- The keyword `cancelbranch` allows you define a “Cancel” branch for aborting the execution of a virtual application. A `cancelbranch` is set to ignore any user input and is implicitly assigned caption `FSCVENV@1.1001:StrCancel` and symbol `COODESK@1.1:SymbolCancel`. Moreover, the default branch expression `coouser.Cancel()` is implicitly assigned to a `cancelbranch`.

The default branch handler expression, caption, and symbol of both `nextbranch` and `cancelbranch` can be overridden using the `expression`, `caption`, and `symbol` keywords.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCVAPP@1.1001;
  import FSCVENV@1.1001;
  menu usecase EditInvoice on selected {
    variant Order {
      impl = application {
        boolean invoiceprinted;
        Invoice invoice;
        expression {
          invoice = cooobj.orderinvoice;
          if (invoice != null) {
            invoiceprinted = false;
            invoice.ObjectLock(true, true);
            ->EditInvoiceDialog;
            if (invoiceprinted) {
              invoice->MailInvoice();
            }
          }
          else {
            throw coort.SetError(#NoInvoiceFound, null);
          }
        }
        dialog EditInvoiceDialog {
          form = PageInvoice;
          target = invoice;-
          branch printinvoice {
            symbol = SymbolPrinter;
            caption = StrPrintInvoice;
            visible = expression {
              invoice.CheckGetAccess(cootx, #content)
            }
            expression {
              try {
                invoice->PrintInvoice();
                invoiceprinted = true;
              }
              catch (@error) {
                if (coort.GetErrorMessage(@error) != #COOSTERR_CANCEL) {
                  throw @error;
                }
              }
            }
          }
          nextbranch;
          behaviors {
            required {
              applytofield = "objname";
            }
            editcondition {
              condition = expression {
                Invoice invoice;
                invoice.invoicepaymentdate == null
              }
              applytofield = "invoicestate";
            }
            required, editcondition, create {
              condition = expression{
                coort.GetCurrentUserRolePosition() == #SysAdm
              }
              applytofield = "invoiceorder";
            }
          }
        }
```

```
      }
    }
  }
}
```

8.3.4 Extending a dialog

Use the `using` keyword followed by the reference of the generated virtual application (for the combination of variant and use case) and curly braces. The virtual application contains the dialog that should be extended. To extend a dialog with a branch, use the `extend dialog` keywords followed by the dialog reference and curly braces.

Syntax

```
using reference {
  extend dialog reference {
    ...
  }
}
```

Cloud profile note: The extension of dialogs that belong to another non-friend software component is not allowed.

In the following example the “EditInvoiceDialog” of the menu use case “EditInvoice” that is defined for variant “Order” gets extended by a cancel branch.

Example

```
using OrderEditInvoice {
  extend dialog EditInvoiceDialog {
    cancelbranch;
  }
}
```

8.4 Implementing a use case in Java

Before you can start to implement use cases in Java, you have to enable Java support for your Fabasoft app.ducx project. You can enable Java support by selecting *Enable Java support* when creating a new Fabasoft app.ducx project. Java support may also be enabled later in the “Properties” dialog box of your Fabasoft app.ducx project (see next figure).

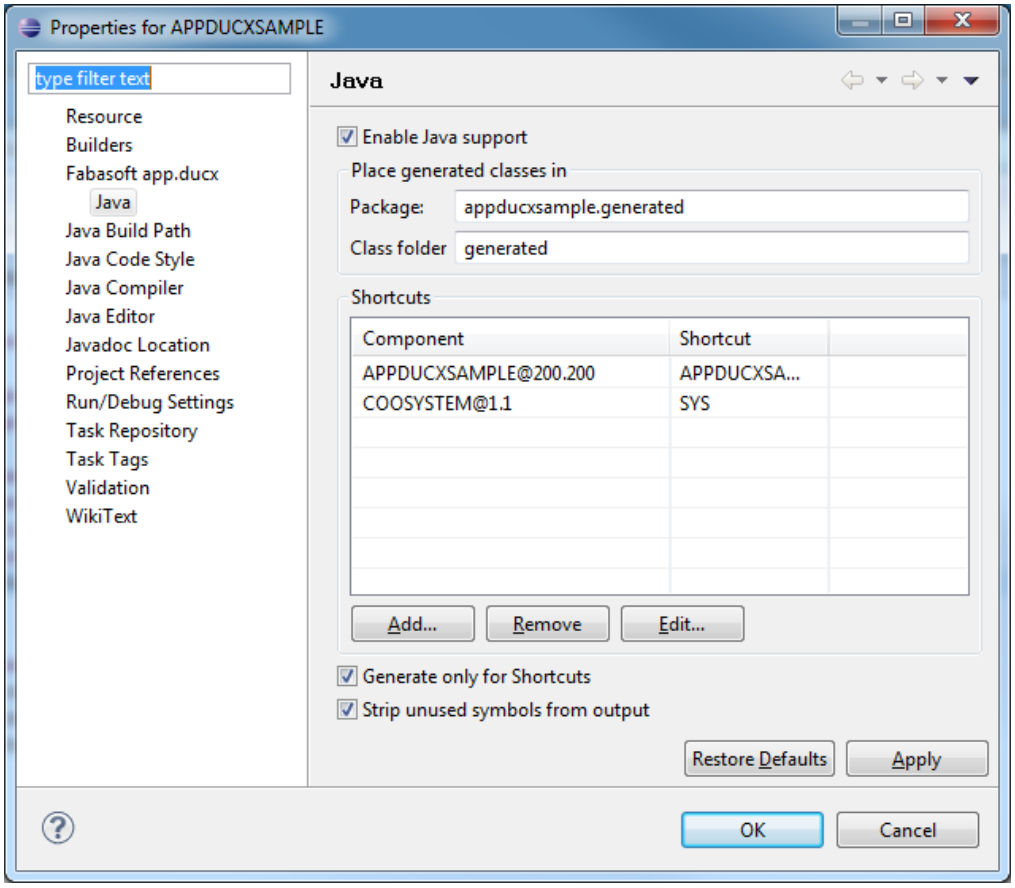


Figure 20: Enabling Java support for a Fabasoft app.ducx project

When enabling Java support, you have to provide a *Package* name and a *Class folder* for the classes generated automatically by Fabasoft app.ducx to allow you to access the Fabasoft Folio object model in Java. These classes are encapsulated in a Java package and put in a class folder, which is then added to your Fabasoft app.ducx project. The files generated by Fabasoft app.ducx are updated automatically whenever you modify your object model or your use case definitions.

It is possible to generate Java classes only for software components that are defined as shortcuts. Shortcuts for `COOSYSTEM@1.1` and the own software component are mandatory and created automatically if not already defined. This way the compiling performance can be improved.

Strip unused symbols from output should be used for a release build to reduce the size of the JAR file. No stripping of unused symbols is an advantage in the development process because the project compiles faster.

Note:

- JAR files of other software components in the Fabasoft Folio Domain are fetched when executing "Update All References". These JAR files can be referenced within the Fabasoft app.ducx project (“Properties” > “Java Build Path” > “Libraries” > “Add JARs”). The JAR files are located in the `.references\lib` folder. In a newly created project it may be necessary to refresh the project to be able to select the JAR files.
  Because the JAR files of the Fabasoft Folio Domain are used, no upload of these files is needed.
- In the following examples, the package name APPDUCXSAMPLE`.generated` is used for the class files generated by Fabasoft app.ducx.

8.4.1 Defining a use case to be implemented in Java

The app.ducx use case language must be used to define the use case. For further information on how to define a use case, please refer to chapter 8.2 “Defining use cases”.

When specifying the name of the Java method containing the implementation of your use case, the following syntax has to be used:

Syntax

```
impl = java:package.class.method;
```

After the `java:` prefix, specify the package name for your Java implementation package followed by the reference of the object class, and the name of the method. The individual parts must be separated by periods.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  usecase CreateInvoice(out Invoice invoice) {
    variant Order {
      impl = java:APPDUCXSAMPLE.Order.CreateInvoice;
    }
  }
}
```

8.4.2 Implementing the Java method

When defining a use case implemented as a Java method, Fabasoft app.ducx will offer to generate a Java class and a method stub for your use case in the Java class on behalf of you.

After providing the name of the java method in the use case file, an error symbol is shown on the left-hand side next to the corresponding line in the source code indicating that the Java method does not exist yet. A Left-click on the error symbol opens a Quick Fix (see next figure).

In the Fabasoft app.ducx wizard dialog box, double-click “Create method…” to let the Fabasoft app.ducx wizard create a Java method stub for your use case.

Java methods may also be defined manually without the help of the app.ducx wizard. However, please note that all Java methods invoked from a use case must be attributed in the following format:

```
@DUCXImplementation("<Fully Qualified Reference of the Use Case>")
```

Furthermore, you may add your own member variables and methods that are not invoked by a use case to the Java classes generated by the Fabasoft app.ducx wizard as well.

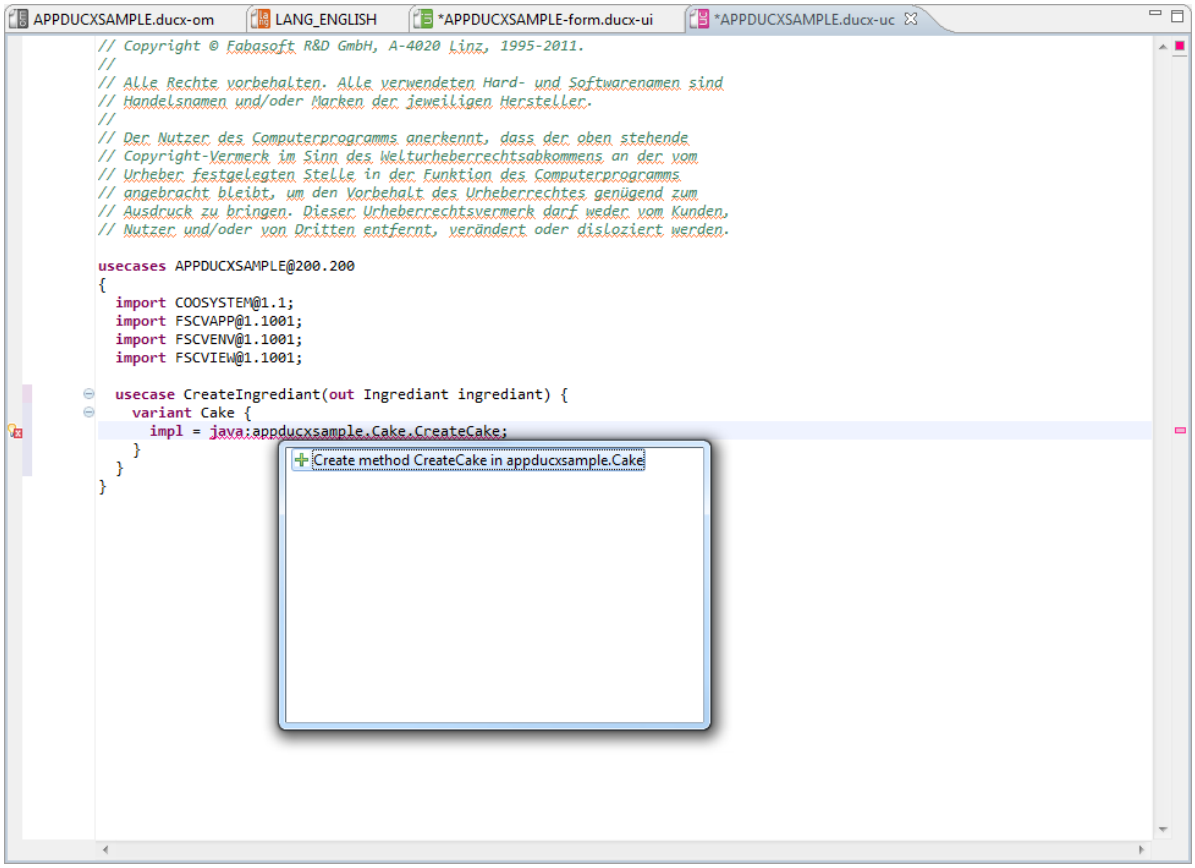


Figure 21: app.ducx wizard for creating a Java method stub

The following example illustrates the Java source code generated by the Fabasoft app.ducx wizard for a typical Java method stub.

Example

```
package APPDUCXSAMPLE;

import CooLib.CooObject;
import CooLib.CooException;
import static APPDUCXSAMPLE.generated.DUCX.coort;
import static APPDUCXSAMPLE.generated.DUCX.cootx;
import APPDUCXSAMPLE.generated.DUCXImplementation;
import APPDUCXSAMPLE.generated.APPDUCXSAMPLE_200_300.actions.CreateInvoiceResult;

public class Order extends APPDUCXSAMPLE.generated.APPDUCXSAMPLE_200_300.
  classes.Order {
  public Order(CooObject obj) {
    super(obj);
  }

  @DUCXImplementation("APPDUCXSAMPLE@200.200:CreateInvoice")
  public CreateInvoiceResult CreateInvoice() throws CooException {
    CreateInvoiceResult result = new CreateInvoiceResult();
    // TODO: Auto-generated implementation stub
    return result;
  }
}
```

For a comprehensive example demonstrating how to implement a use case in Java, please refer to chapter 14.1 “Comprehensive Java example”.

Note: The examples in the remainder of this chapter are not self-contained in order to improve readability. Required import statements are omitted to keep the examples more concise.

8.4.3 Importing packages generated by Fabasoft app.ducx

The classes generated by Fabasoft app.ducx allow you to access the Fabasoft Folio object model so you can retrieve and set property values, invoke other use cases, and access component objects such as object classes. These classes reside in a separate package. The name of this package must be specified along with the class folder when enabling Java support for your app.ducx project.

Fabasoft app.ducx generates a set of Java class files for all software components referenced in your Fabasoft app.ducx project. The generated Java class files are updated as you add or remove software component references.

For each software component referenced in your Fabasoft app.ducx project, a package is created and the reference of the software component is used as package name. In this package, a class with the same name as the package is provided which contains get methods for accessing all component objects that belong to the software component.

This package also contains four sub packages that allow you to access object classes, to invoke use cases, and to use enumeration types and compound types:

- The `classes` package contains classes for accessing object classes.
- The `actions` package contains classes for transporting the output parameters when invoking use cases.
- The `enums` package contains classes for working with enumeration types.
- The `structs` package contains classes for working with compound types.

In order to keep your source code concise, it is recommended to add import statements to the header of your source files to import the packages you need.

Example

```
// Import for accessing component objects belonging to APPDUCXSAMPLE@200.200,
// e.g. the object representing the software component
import APPDUCXSAMPLE.generated.APPDUCXSAMPLE_200_300.APPDUCXSAMPLE_200_300;

// Import for making available object class APPDUCXSAMPLE@200.200:Invoice
import APPDUCXSAMPLE.generated.APPDUCXSAMPLE_200_300.classes.Invoice;

// Import for making available class CreateInvoiceResult for retrieving
// the output parameters after invoking the use case APPDUCXSAMPLE@200.200:
// CreateInvoice
import APPDUCXSAMPLE.generated.APPDUCXSAMPLE_200_300.actions.CreateInvoiceResult;

// Import for making available the enumeration type APPDUCXSAMPLE@200.200:
// InvoiceState
import APPDUCXSAMPLE.generated.APPDUCXSAMPLE_200_300_enums.InvoiceState;

// Import for making available the compound type APPDUCXSAMPLE@200.200:
// OrderPosition
import APPDUCXSAMPLE.generated.APPDUCXSAMPLE_200_300.structs.OrderPosition;
```

In addition to the imports of packages generated by Fabasoft app.ducx, you also need to import parts of the `CooLib` package which contains some required classes. The library containing the `CooLib` package is automatically added to your project.

8.4.4 Data types in Java

Table 25 contains the list of data types used for accessing the Fabasoft Folio object model in Java.

Fabasoft Folio Data Type	app.ducx Keyword	Java Class
`COOSYSTEM@1.1:STRING`	`string`	`java.lang.String`

COOSYSTEM@1.1:BOOLEAN	boolean	java.lang.Boolean
COOSYSTEM@1.1:INTEGER	integer, time, timespan	java.lang.Integer
COOSYSTEM@1.1:FLOAT	float	java.lang.Double
COOSYSTEM@1.1:DATETIME	date, datetime	java.util.Date
COOSYSTEM@1.1:Currency	currency	COOSYSTEM_1_1.structs.Currency
COOSYSTEM@1.1:CONTENT	content	CooLib.CooContent
COOSYSTEM@1.1:DICTIONARY	dictionary	CooLib.CooDictionary
COOSYSTEM@1.1:OBJECT	object	CooLib.CooObject

Table 25: Data types in Java

Typed lists are used for accessing lists of values, i.e. properties allowing multiple values. For instance, the `List<String>` interface can be used for accessing a string list property of data type `COOSYSTEM@1.1:STRINGLIST`.

The type definition component objects for simple data types are accessible over the `app.ducx` class. For example, `DUCX.getTypeBoolean()` returns the type definition component object `COOSYSTEM@1.1:BOOLEAN`.

8.4.5 Accessing properties and invoking use cases

The Fabasoft app.ducx wizard automatically generates methods for accessing properties and use cases belonging to the software components referenced in your Fabasoft app.ducx project. The generated methods allow you to retrieve and set property values and to invoke use cases.

Note: The object the use case is invoked on can be accessed using the `this` variable.

Please refer to chapter 8.4.6 “Working with use case parameters” for detailed information on how to handle input and output parameters when implementing or invoking use cases.

Example

```
@DUCXImplementation("APPDUCXSAMPLE@200.200:InitializeOrder")
public void InitializeOrder(final Person customer) throws CooException {
  // Lock the order object
  this.COOSYSTEM_1_1_ObjectLock(true, true, null, null);
  // Initialize order date with current date and set order status to
  // OS_PENDING
  this.APPDUCXSAMPLE_200_300_orderdate = new Date();
  this.APPDUCXSAMPLE_200_300_orderstate = OrderState.OS_PENDING;
  // Set express delivery for wholesalers
  if (customer.APPDUCXSAMPLE_200_300_customertype == CustomerType.
    CT_WHOLESALER) {
    this.APPDUCXSAMPLE_200_300_orderdelivery = DeliveryType.DT_EXPRESS;
  }
  else {
    this.APPDUCXSAMPLE_200_300_orderdelivery = DeliveryType.DT_STANDARD;
  }
  // Add the order to the customer's list of orders
  this.COODESK_1_1_ShareObject(null, null, APPDUCXSAMPLE_200_300.
    getProperty customerorders(), customer);
}
```

8.4.6 Working with use case parameters

8.4.6.1 Retrieving input parameters

When you add a method implementation to your Java project, a signature is generated automatically for the new Java method. This signature contains a parameter list that corresponds to the parameter list of your use case. The input parameters defined for the use case are passed in to your Java method as Java objects.

Example

app.ducx Use Case Language

```
usecase AddPositionToOrder(Product product, integer quantity) {
  variant Order {
    impl = java:APPDUCXSAMPLE.Order.AddPositionToOrder;
  }
}
```

Java Implementation

```
@DUCXImplementation("APPDUCXSAMPLE@200.200:AddPositionToOrder")
public void AddPositionToOrder(final Product product, final Long
  quantity) throws CooException {
  // Lock the order
  this.COOSYSTEM_1_1_ObjectLock(true, true, null, null);
  // Create a new order position
  OrderPosition position = OrderPosition.create();
  position.APPDUCXSAMPLE_200_300_product = product;
  position.APPDUCXSAMPLE_200_300_quantity = quantity;
  // Add the new order position to the order
  List<OrderPosition> positions = this.APPDUCXSAMPLE_200_300_orderpositions;
  positions.add(position);
  this.APPDUCXSAMPLE_200_300_orderpositions = positions;
}
```

8.4.6.2 Returning output parameters

Java methods do not support output parameters. Instead, a so-called result object must be used to return output parameters. The corresponding Java class for creating a result object is automatically created by a wizard when adding a new use case implementation in Java if the use case has one or more output parameters. The name of the Java class for retrieving the output parameters corresponds to the reference of the use case to be invoked, suffixed by the string “Result”.

When invoking another use case from within your Java method, you can follow the same approach for retrieving its output parameters: the invoked use case returns a result object containing a member variable for each output parameter.

Example

app.ducx Use Case Language

```
usecase GetCustomerOrders(out Order[] orders) {
  variant Person {
    impl = java:APPDUCXSAMPLE.Person.GetCustomerOrders;
  }
}
```

Java Implementation

```
@DUCXImplementation("APPDUCXSAMPLE@200.200:GetCustomerOrders")
public GetCustomerOrdersResult GetCustomerOrders() throws CooException {
  // Output parameters must be passed back using the result object
  GetCustomerOrdersResult result = new GetCustomerOrdersResult();
  // The sort order is determined by a list of property definitions
  ArrayList<Object> sortorder = new ArrayList<Object>();
```

```
  sortorder.add(APPDUCXSAMPLE_200_300.getProperty_orderdate());
  // Sort the customer's list of orders (the sorted list is returned in
  // the valuelist member variable of the SortResult object)
  SortResult sortresult = this.COOATTREDIT_1_1_Sort(
    this.APPDUCXSAMPLE_200_300_customerorders, true, sortorder, null);
  // Return the sorted list of orders
  result.orders = sortresult.valuelist;
  return result;
}
```

The result object provides the two methods `SetError(CooObject)` and `SetError(CooObject, String)`. When calling `SetError` the output parameters are saved and afterwards the error raises an exception.

8.4.7 Working with objects

8.4.7.1 Creating new objects in Fabasoft Folio

For creating a new instance of an object class, you can invoke the `create` method of the Java class representing the object class.

In the following example, a new instance of object class `APPDUCXSAMPLE@200.200:Invoice` is created and referenced in the `APPDUCXSAMPLE@200.200:orderinvoice` property of an order.

Example

```
@DUCXImplementation("APPDUCXSAMPLE@200.200:MarkOrderAsShipped")
public void MarkOrderAsShipped() throws CooException {
  // Lock the order object
  this.COOSYSTEM_1_1_ObjectLock(true, true, null, null);
  // Create a new instance of object class APPDUCXSAMPLE@200.200:Invoice
  Invoice invoice = Invoice.create();
  // Reference the new invoice object in the order object and set
  // the order's state to OS_SHIPPED
  this.APPDUCXSAMPLE_200_300_orderinvoice = invoice;
  this.APPDUCXSAMPLE_200_300_orderstate = OrderState.OS_SHIPPED;
}
```

8.4.7.2 Comparing objects

You can use the `equals` method of an object to determine if it is equivalent to a given object instance in Fabasoft Folio.

Example

```
@DUCXImplementation("APPDUCXSAMPLE@200.200:ProcessPendingOrders")
public void AddOrderCategory(final Case case) throws CooException {
  ComponentDocumentCategory category = this.COOTC_1_1001_objcategory;
  List<Object> casecategories = case.COOTC_1_1001_allowedcategories;
  // Check whether APPDUCXSAMPLE@200.200:DocumentCategoryOrder was selected
  // for the order object
  if (category.equals(APPDUCXSAMPLE_200_300.getClass_DocumentCategoryOrder())) {
    if (!casecategories.contains(APPDUCXSAMPLE_200_300.
      getClass_DocumentCategoryOrder())) {
      ComponentDocumentCategory.from(APPDUCXSAMPLE_200_300.
        get_DocumentCategoryOrder()).COODESK_1_1_ShareObject(null, null,
        COOTC_1_1001.getProperty_allowedcategories(), case);
    }
    else {
      coort.SetError(APPDUCXSAMPLE_200_300.
      getErrorMessage_CategoryAlreadyAdded());
    }
```

```
  }
  else {
    coort.SetError(APPDUCXSAMPLE_200_300.
    getErrorMessage UnexpectedOrderCategory());
  }
}
```

8.4.7.3 Important methods of an object

Table 26 summarizes the most important methods that can be invoked on instances of Fabasoft Folio objects.

Method	Description
`isValid()`	`isValid` determines whether the object is valid.
`IsClass(class)`	`IsClass` determines whether the object class it is invoked on is derived from or identical to `class`.
`HasClass(class)`	`HasClass` determines whether the object is an instance of or derived from `class`.
`GetClass()`	`GetClass` returns the object class of the object.
`GetName()`	`GetName` returns the *Name* (`COOSYSTEM@1.1:objname`) of the object. If the name of the object cannot be read (e.g. because the user does not have sufficient rights for reading the object name), a string explaining why the name of the object cannot be read is returned.
`GetAddress()`	`GetAddress` returns the *Address* (`COOSYSTEM@1.1:objaddress`) of the object.
`GetReference()`	`GetReference` returns the *Reference* (`COOSYSTEM@1.1:reference`) of a component object.
`GetIdentification()`	`GetIdentification` returns the full identification of the object, which is a combination of the *Address* (`COOSYSTEM@1.1:objaddress`) and a version timestamp.
`GetVersNr()`	`GetVersNr` returns the version number of the object. If no version exists, „0“ is returned.
`GetVersDate()`	`GetVersDate` returns the date and time of the version of the object. This method can only be used for versions of objects, otherwise an exception is thrown. Use `GetVersNr` to find out whether the object is a version object.
`LoadAllAttributes(tx)`	`LoadAllAttributes` loads the values of all properties into the Fabasoft Folio Kernel Cache.
`LoadSpecificAttributes(tx, properties)`	`LoadSpecificAttributes` loads the values of the properties specified in `properties` into the Fabasoft Folio Kernel Cache.
`HasAttribute(tx, property)`	`HasAttribute` checks whether the specified `property` is assigned

	to the object.
`HasAttributeValue(tx, property)`	`HasAttributeValue` checks whether the specified `property` is assigned to the object and whether it is assigned a value.
`CheckAccess(tx, accesstype)`	`CheckAccess` checks whether accessing the object with the specified access type is allowed.
`CheckGetAccess(tx, property)`	`CheckGetAccess` checks whether the specified `property` of the object may be read.
`CheckSetAccess(tx, property)`	`CheckGetAccess` checks whether the specified `property` of the object may be changed.
`IsGhost()`	`IsGhost` checks whether object is still alive.

Table 26: Object methods

8.4.8 Working with enumeration types, compound types, contents, and dictionaries

8.4.8.1 Working with enumeration types

The app.ducx wizard automatically generates Java classes for each enumeration type. These classes allow you to access all enumeration items of a particular enumeration type.

Example

```
@DUCXImplementation("APPDUCXSAMPLE@200.200:ArchiveOrder")
public void ArchiveOrder() throws CooException {
  if (this.APPDUCXSAMPLE_200_300_orderstate == OrderState.OS_COMPLETED) {
    // Lock the order object and set the order state to OS ARCHIVED
    this.COOSYSTEM_1_1_ObjectLock(true, true, null, null);
    this.APPDUCXSAMPLE_200_300_orderstate = OrderState.OS_ARCHIVED;
    // Invoke an XML web service to archive the order
    archiveOrderXML();
  }
  else {
    // Raise an error if the order is not completed yet
    coort.SetError(APPDUCXSAMPLE_200_300.getErrorMessage_InvalidOrderState());
  }
}
```

8.4.8.2 Working with compound types

Data structures representing compound types are referred to as compound values. Compound values must be created before they can be used. The `create` method of the Java class representing the compound type must be invoked for creating a new compound value.

The following example demonstrates how to add a new item to a compound property.

Example

```
@DUCXImplementation("APPDUCXSAMPLE@200.200:AddPositionToOrder")
public void AddPositionToOrder(final Product product,
  final Long quantity) throws CooException {
  // Lock the order object
  this.COOSYSTEM_1_1_ObjectLock(true, true, null, null);
  // Retrieve the existing order positions
  List<OrderPosition> positions = this.APPDUCXSAMPLE_200_300_orderpositions;
  // Create a new entry and add it to the list of existing order positions
```

```
  OrderPosition position = OrderPosition.create();
  position.APPDUCXSAMPLE_200_300_product = product;
  position.APPDUCXSAMPLE_200_300_quantity = quantity;
  positions.add(position);
  // Write the modified list of order positions back to the order
  this.APPDUCXSAMPLE_200_300_orderpositions = positions;
}
```

8.4.8.3 Working with contents

In Java, contents are represented by the `CooContent` class. The next table shows the most important methods supported by `CooContent`.

Method	Description
`GetFile(name, generatetemp)`	`GetFile` copies the `CooContent` to a file and returns the file name. `name` denotes the name of the file to be created.
`SetFile(name, removeonrelease)`	`SetFile` stores the content of a file in a `CooContent`. `name` denotes the name of the file to be used as a source.
`GetContent(tx, flags, codepage)`	`GetContent` returns a `CooContent` as a `String` and can only be used for retrieving text-based contents.
`SetContent(tx, flags, codepage, string)`	`SetContent` stores a `String` in a `CooContent`.

Table 27: Methods of the CooContent class

In the following example, a string property of type `COOSYSTEM@1.1:STRINGLIST` is retrieved as a scalar string and stored in a `CooContent`. Please note that the `CooContent` has to be initialized first by invoking the `CreateContent` method of the Fabasoft Folio Runtime. Afterwards, the `CooContent` is stored in a compound value of type `COOSYSTEM@1.1:Content`.

Example

```
@DUCXImplementation("APPDUCXSAMPLE@200.200:SendProductDescription")
public void SendProductDescription() throws CooException {
  // Get the product description as a scalar string value
  // Note: this.APPDUCXSAMPLE_200_300_productdescription returns a List<String>
  String description = this.GetAttributeString(cootx, APPDUCXSAMPLE_200_300.
    getProperty_productdescription(), null, CooFlags.COODISP_ROWLIST);
  // Store the description string in a content
  CooContent descriptioncontent = coort.CreateContent();
  descriptioncontent.SetContent(cootx, CooFlags.COOGC_MULTIBYTEFILE,
    CooFlags.COOGC_UTF8, description);
  // Create a structure of compound type COOSYSTEM@1.1:Content and
  // store the content in this structure
  Content content = Content.create();
  content.COOSYSTEM_1_1_contcontent = descriptioncontent;
  content.COOSYSTEM_1_1_contextension = "txt";
  // Invoke an XML web service to transmit the content
  sendProductDescriptionXML(content);
}
```

8.4.8.4 Working with dictionaries

In Java, dictionaries are represented by the `CooDictionary` class, which allows you to store a list of key-value pairs. The key must be a `String` value. The next table lists the most important methods supported by the `CooDictionary` class.

Method	Description
`GetEntry(key)`	`GetEntry` returns the list of values stored under `key`. Use this method for retrieving lists from a dictionary.
`GetEntryValue(key)`	`GetEntryValue` returns value stored under `key`. Use this method for retrieving scalar values from a dictionary.
`GetEntryValueCount(key)`	`GetEntryValueCount` returns the number of values of the entry specified by `key`.
`GetEntryCount()`	`GetEntryCount` returns the number of entries in a dictionary.
`GetEntryKey(index)`	`GetEntryKey` returns the key of the entry of the specified `index`.
`SetEntry(key, values)`	`SetEntry` creates an entry under `key` for the specified `values`. Use this method for storing lists of values in a dictionary.
`SetEntryValue(key, value)`	`SetEntryValue` creates an entry under `key` for the specified `value`. Use this method for storing a scalar value in a dictionary.
`TestEntry(key)`	`TestEntry` checks whether a dictionary contains an entry under `key`. This method returns `true` if the value stored under `key` is `null`.
`HasEntry(key)`	`HasEntry` checks whether a dictionary contains an entry under `key`. This method returns `false` if the value stored under `key` is `null`.
`ClearEntry(key)`	`ClearEntry` removes the entry stored under `key` from a dictionary.
`Reset()`	`Reset` removes all entries from a dictionary.
`Backup()`	`Backup` serializes the contents of a dictionary to a string.
`Restore(string)`	`Restore` rebuilds a dictionary from a serialized `string`.

Table 28: Methods of the CooDictionary class

In many cases, dictionaries are used as local or global scope when evaluating app.ducx expressions.

Note: Before you can use a `CooDictionary`, it has to be initialized by invoking the `CreateDictionary` method of the Fabasoft Folio Runtime.

In the following example, an expression is evaluated by the implementation of use case `APPDUCXSAMPLE@200.200:ValidateInvoice`. The invoice object is provided in the local scope, and a dictionary is provided in the global scope.

Example

```
@DUCXImplementation("APPDUCXSAMPLE@200.200:ValidateInvoice")
public void ValidateInvoice() throws CooException {
  Invoice invoice = this;
  Date paymentdate = invoice.APPDUCXSAMPLE 200 300 invoicepaymentdate;
  InvoiceState invoicestate = invoice.APPDUCXSAMPLE_200_300_invoicestate;
  String expressiontext = getValidationExpression(invoice);
  // Create and populate global scope dictionary
  CooDictionary globalscope = coort.CreateDictionary();
  globalscope.SetEntryValue("paydate", paymentdate);
  globalscope.SetEntryValue("state", invoicestate);
  // Evaluate validation expression and get result
  CooValue[] result = CooExpression.Evaluate(cootx, expressiontext,
    CooValue.asArray(globalscope), CooValue.asArray(invoice));
  if (result[0].getBool().getValue()) {
    // Validation expression returned "true"
    processInvoice(invoice);
  }
  else {
    // Validation expression returned "false"
    coort.SetError(APPDUCXSAMPLE_200_300.getErrorMessage_ValidationError());
  }
}
```

8.4.9 Accessing the Fabasoft Folio Runtime

When implementing a method in Java, the static variable `coort` of the `DUCX` package can be used to access the Fabasoft Folio Runtime. The next table contains a list of the most important methods supported by the Fabasoft Folio Runtime.

Method	Description
`CreateContent()`	`CreateContent` initializes a new `CooContent` object. Please note that this is a memory structure, and not a persistent object in Fabasoft Folio.
`CreateDictionary()`	`CreateDictionary` initializes a new `CooDictionary` object. Please note that this is a memory structure, and not a persistent object in Fabasoft Folio.
`GetCurrentUser()`	`GetCurrentUser` returns the user object of the user currently logged in to Fabasoft Folio.
`GetCurrentUserEnvironment()`	`GetCurrentUserEnvironment` returns the active user environment object of the user currently logged in to Fabasoft Folio.
`GetCurrentUserRoot()`	`GetCurrentUserRoot` returns the desk object of the user currently logged in to Fabasoft Folio.
`GetCurrentUserLanguage()`	`GetCurrentUserLanguage` returns the language of the user currently logged in to Fabasoft Folio.
`GetCurrentUserRoleGroup()`	`GetCurrentUserRoleGroup` returns the group object of the current role of the user currently logged in to Fabasoft Folio.
`GetCurrentUserRolePosition()`	`GetCurrentUserRolePosition` returns the position object of

	the current role of the user currently logged in to Fabasoft Folio.
`GetObject(address)`	`GetObject` returns the object with the specified `address`.
`SearchObjects(tx, query)`	`SearchObjects` executes a Fabasoft Folio `query` and returns the matching objects.
`SetError(errormessage)`	`SetError` returns a `CooException` using the `errormessage` object.
`LoadAllAttributes(tx, objects)`	`LoadAllAttributes` loads the values of all properties of the specified `objects` into the Fabasoft Folio Kernel Cache.
`Trace(string)`	`Trace` writes the specified `string` to the Fabasoft app.ducx Tracer (also if the software component is not in trace mode).
`Trace(string, value)`	`Trace` writes the specified `string` and a value of arbitrary data type to the Fabasoft app.ducx Tracer (also if the software component is not in trace mode).
`ReportEvent(source, string, type, category)`	`ReportEvent` creates a new entry in the event log.

Table 29: Methods of the Fabasoft Folio Runtime

Example

```
@DUCXImplementation("APPDUCXSAMPLE@200.200:ProcessPendingOrders")
public void ProcessPendingOrders() throws CooException {
  // Get the pending orders of the current user
  User user = (User) coort.GetCurrentUser();
  List<Order> pendinglist = user.APPDUCXSAMPLE_200_300_pendingorders;
  // Trace the number of pending orders
  coort.Trace("Found " + pendinglist.size() + " pending orders");
  // Load all properties of the pending order objects
  Order[] orders = pendingList.toArray(new Order[pendinglist.size()]);
  coort.LoadAllAttributes(cootx, orders);
  processPendingOrders(orders);
}
```

8.4.10 Accessing the transaction context

When implementing a method in Java, the static variable `cootx` of the `DUCX` package can be used to access the current transaction context. `cootx` returns an instance of Java class `CooTransaction`. The most important methods of the `CooTransaction` class are listed in the next table.

Method	Description
`Abort()`	`Abort` aborts the current transaction and rolls back any changes.
`Commit()`	`Commit` closes the current transaction and stores any changes.
`CommitEx(flags)`	`CommitEx` closes the current transaction and stores any

	changes.
	Additionally, the following flags are supported:
	`COOCF_NORMAL`
	Normal commit
	`COOCF_KEEPONFAILURE`
	If commit fails, all transaction data is kept
	`COOCF_KEEPREFRESHINFO`
	After commit the refresh info of all touched object is kept
	`COOCF_KEEPSEARCHINFO`
	After commit the search info of the searches executed in this transaction is kept
	`COOCF_KEEPVARIABLES`
	After commit all transaction variables are kept
	`COOCF_KEEPLOCKS`
	After commit all locks of objects are kept
	`COOCF_KEEPOBJECTS`
	After commit all modified objects are stored in the transaction variable `COOSYSTEM@1.1:TV_COMMITTEDOBJECTS`
	`COOCF_NOTELESS`
	If specified the properties `COOSYSTEM@1.1:objmodifiedat` and `COOSYSTEM@1.1:objchangedby` are not set. This flag is only allowed, if the current user has the role `COOSYSTEM@1.1:SysAdm` and the current user is registered in `COOSYSTEM@1.1:domainmasterusers` of the current domain object

`Persist(object)`	`Persist` temporarily stores the state of the current transaction without committing the changes.
`Clone()`	`Clone` returns a clone of the current transaction.
`HasVariable(swc, id)`	`HasVariable` checks whether transaction variable `id` of software component `swc` contains a value. This method returns `false` if if the value stored is `null`.
`TestVariable(swc, id)`	`TestVariable` checks whether transaction variable `id` of software component `swc` contains a value. This method returns `true` even if the value stored is `null`.
`ClearVariable(swc, id)`	`ClearVariable` removes transaction variable `id` of software component `swc` from the transaction.
`GetVariable(swc, id)`	`GetVariable` retrieves the list of values stored in transaction variable `id` of software component `swc`.
`SetVariable(swc, id, type, values)`	`SetVariable` stores the specified `values` in transaction variable `id` of software component `swc`.

`GetVariableValueCount(swc, id)`	`GetVariableValueCount` returns the number of values stored in transaction variable `id` of software component `swc`.
`HasVariableValue(swc, id)`	`HasVariableValue` returns `true` if a transaction variable `id` of software component `swc` is available.
`GetVariableValue(swc, id)`	`GetVariableValue` retrieves a scalar `value` stored in transaction variable `id` of software component `swc`.
`SetVariableValue(swc, id, type, value)`	`SetVariableValue` stores the specified scalar `value` in transaction variable `id` of software component `swc`.
`GetVariableString`	
`GetVariableStringEx`	
`GetVariableTypeDefinition(swc, id)`	`GetVariableTypeDefinition` returns the type definition for the variable stored in transaction variable `id` of software component `swc`.
`IsClone()`	
`IsModified()`	`IsModified` checks whether objects were modified within the transaction.
`IsModifiedEx()`	`IsModifiedEx` checks whether any data was modified within the transaction.
`IsCreated(object)`	`IsCreated` checks whether `object` was created in this transaction.
`IsDeleted(object)`	`IsDeleted` checks whether `object` was deleted in this transaction.
`IsChanged(object)`	`IsChanged` checks whether `object` was changed in this transaction.
`IsAttributeChanged(object, property)`	`IsAttributeChanged` checks whether `property` of `object` was changed in this transaction.

`GetTransactionFlags()`	`GetTransactionFlags` retrieves the flags of the transaction:
	`COOTXF_ROLECHANGED`
	During the transaction an automatic role change has been performed
	`COOTXF_NOREFRESH`
	Objects are not automatically refreshed when accessed with this transaction
	`COOTXF_NOAUTOVERSION`
	During commit of the transaction no automatic version will be created

SetTransactionFlags(flags)	
Backup()	
Restore(data)	
OpenScope()	
CloseScope()	
GetMaster()	

Table 30: Methods of the CooTransaction class

Example

```
@DUCXImplementation("COOSYSTEM@1.1:ObjectPrepareCommit")
public void InvoicePrepareCommit(final Boolean internalchange)
  throws CooException {
  // Check if the invoice has been created in the current transaction
  // and if property APPDUCXSAMPLE@200.200:invoicestate has been changed
  if (!cootx.IsCreated(this) && cootx.IsAttributeChanged(this,
    APPDUCXSAMPLE_200_300.getProperty_invoicestate())) {
    // Rebuild the object's name based on the name build configuration
    this.FSCCONFIG_1_1001_ObjectPrepareCommitNameBuild();
  }
}
```

8.4.10.1 Creating a new transaction

In some scenarios it is necessary to carry out operations in a separate transaction. Any changes that have been made in a new transaction can be committed or rolled back separately from the main transaction.

Example

```
@DUCXImplementation("APPDUCXSAMPLE@200.200:CreateInvoice")
public void CreateInvoice() throws CooException {
  // Create a new transaction
  CooTransaction backuptx = DUCX.cootx;
  CooTransaction localtx = new CooTransaction();
  try {
    // Perform the following operations in context of the new transaction
    coort.SetThreadTransaction(localtx);
    Invoice invoice = Invoice.create();
    this.APPDUCXSAMPLE_200_300_InitializeInvoice(invoice);
    // Commit the changes
    localtx.Commit();
  }
  catch (Exception ex) {
    // In case of an error, roll back the changes
    localtx.Abort();
  }
  finally {
    // Restore original transaction context
    coort.SetThreadTransaction(backuptx);
  }
}
```

8.4.10.2 Working with transaction variables

A transaction variable is a temporary variable identified by a software component, a transaction variable reference, and a unique identification number. Transaction variables must be declared using the app.ducx use case language before they can be used. For further information on how to declare transaction variables, please refer to chapter 8.1 “Declaring transaction variables”.

For accessing a transaction variable, you need a combination of either software component reference and transaction variable reference or software component reference and transaction variable identification number.

Transaction variables are stored and transported in the so called transaction context, i.e. they are cleared when a transaction is committed or aborted.

The purpose of transaction variables is to transfer status information between different use cases invoked within the same transaction.

In some cases, necessary context information cannot be provided in form of parameters when a use case is invoked. For example, the Fabasoft Folio workflow engine makes available a set of four transaction variables (see next table) when a work item of an activity is executed by a user.

Identifier	Description
WFVAR_THIS	Transaction variable WFVAR_THIS of COOWF@1.1 holds the business object attached to the process.
WFVAR_PROCESS	Transaction variable WFVAR_PROCESS of COOWF@1.1 holds the process instance.
WFVAR_ACTIVITY	Transaction variable WFVAR_ACTIVITY of COOWF@1.1 holds the current activity instance.
WFVAR_WORKITEM	Transaction variable WFVAR_WORKITEM of COOWF@1.1 holds the zero-based index of the work item executed by the user.

Table 31: Transaction variables provided by COOWF@1.1 when executing a work item

To access transaction variables, you need to use the access methods exposed by the cootx interface object of the DUCX package (see chapter 8.4.10 “Accessing the transaction context”).

Example

```
@DUCXImplementation("APPDUCXSAMPLE@200.200:ReleaseOrder")
public void ReleaseOrder() throws CooException {
  // Get the order object from transaction variable 1 of COOWF@1.1
  Order order = Order.from(cootx.GetVariableValue(COOWF_1_1.
    getSoftwareComponent(), 1).getObject());
  // Determine if batch mode is enabled by checking transaction
  // variable TV BATCHMODE of COOSYSTEM@1.1
  Boolean batchmode = COOSYSTEM_1_1.getTransactionVariables_TV().
    COOSYSTEM_1_1_TV_BATCHMODE;
  if (order != null && !batchmode) {
    releaseOrder(order);
  }
}
```

For your software component, you may also declare your set of transaction variables using the app.ducx use case language and use them in your Java code as illustrated by the following example.

Example

```
@DUCXImplementation("APPDUCXSAMPLE@200.200:ProcessOrder")
public void ProcessOrder() throws CooException {
  if (processOrder(this)) {
    Invoice invoice = this.APPDUCXSAMPLE_200_300_orderinvoice;
    Boolean printinvoice = invoice.APPDUCXSAMPLE 200 300 invoicestate ==
      InvoiceState.IS_PENDING;
    // Set transaction variables TV_PRINTINVOICE and TV_INVOICE of
    // APPDUCXSAMPLE@200.200
    COOSYSTEM 1 1.getTransactionVariables TV().
      APPDUCXSAMPLE 200 300 TV PRINTINVOICE = printinvoice;
    COOSYSTEM 1 1.getTransactionVariables TV().
      APPDUCXSAMPLE_200_300_TV_INVOICE = invoice;
  }
}
@DUCXImplementation("APPDUCXSAMPLE@200.200:ShipOrder")
public void ShipOrder() throws CooException {
  // Retrieve and evaluate transaction variable TV_PRINTINVOICE of
  // APPDUCXSAMPLE@200.200
  Boolean printinvoice = COOSYSTEM 1 1.getTransactionVariables TV().
    APPDUCXSAMPLE_200_300_TV_PRINTINVOICE
  if (printinvoice) {
    Invoice invoice = COOSYSTEM_1_1.getTransactionVariables_TV().
      APPDUCXSAMPLE_200_300_TV_INVOICE;
    invoice.APPDUCXSAMPLE_200_300_SendInvoice();
  }
  shipOrder(this);
}
@DUCXImplementation("APPDUCXSAMPLE@200.200:SendInvoice")
public void SendInvoice() throws CooException {
  Invoice invoice = this;
  Order order = invoice.APPDUCXSAMPLE 200 300 invoiceorder;
  ContactPerson customer = order.APPDUCXSAMPLE 200 300 ordercustomer;
  CooContent mailbodycontent = coort.CreateContent();
  // Create and populate global scope dictionary
  CooDictionary globalscope = coort.CreateDictionary();
  globalscope.SetEntryValue("invoice", invoice);
  globalscope.SetEntryValue("order", order);
  globalscope.SetEntryValue("customer", customer);
  // Make available the values stored in the global scope dictionary
  // when the invoice report is evaluated by initializing transaction
  // variable 1 of COOAR@1.1
  cootx.SetVariableValue(COOAR_1_1.getSoftwareComponent(), 1,
    DUCX.getTypeDictionary(), globalscope);
  // Evaluate the invoice report template to get the HTML mail body
  CreateReportResult reportresult = invoice.COOAR_1_1_CreateReport(
    APPDUCXSAMPLE_200_300.getClass_InvoiceTemplateReport(), false, null, null,
    mailbodycontent);
  mailbodycontent = reportresult.resultcontent;
  if (mailbodycontent != null) {
    Addressee mainaddress = customer.FSCFOLIO 1 1001 mainaddress;
    String recipient = mainaddress.FSCFOLIO_1_1001_emailaddress;
    String sender = "orderprocessing@APPDUCXSAMPLE.fabasoft.com";
    String subject = invoice.GetName();
    ArrayList<String> recipients = new ArrayList<String>();
    recipients.add(recipient);
    ArrayList<CooContent> mailbody = new ArrayList<CooContent>();
    mailbody.add(mailbodycontent);
    %%TRACE("Mail Recipient:", recipient);
    %%TRACE("Mail Subject:", subject);
    %%TRACE("Mail Body:", mailbodycontent.GetContent(cootx,
      CooFlags.COOGC_MULTIBYTEFILE, CooFlags.COOGC_UTF8));
    // Send the invoice mail to the customer
    invoice.FSCSMTP_1_1001_SendHTML(sender, recipients, subject, mailbody,
      null, null, null, null, null, null, null);
  }
}
```

8.4.11 Tracing in Java

In addition to using the tracing functionality built into Fabasoft app.ducx described in chapter 13.1 “Tracing in Fabasoft app.ducx projects”, you may also write custom trace messages to Fabasoft app.ducx Tracer.

You can use the `Trace` method of the `coort` interface object for invoking the trace functionality of the Fabasoft Folio Runtime. However, this method only supports tracing simple strings.

The `trace` method of the `DUCX` class is more powerful, and allows you to trace all kinds of Fabasoft Folio data types.

Moreover, the `DUCX` class also exposes the `traceEnter` and `traceLeave` methods that can be used for logging in the trace output when your Java method has been entered and left.

Example

```
@DUCXImplementation("APPDUCXSAMPLE@200.200:ProcessPendingOrders")
public void ProcessPendingOrders() throws CooException {
  DUCX.traceEnter();
  // Get the pending orders of the current user
  User user = (User) coort.GetCurrentUser();
  List<Order> pendinglist = user.APPDUCXSAMPLE_200_300_pendingorders;
  // Trace the list of pending orders
  DUCX.trace("List of Orders:", pendinglist);
  // Load all properties of the pending order objects
  Order[] orders = pendingList.toArray(new Order[pendinglist.size()]);
  coort.LoadAllAttributes(cootx, orders);
  processPendingOrders(orders);
  DUCX.traceLeave();
}
```

8.4.12 Support of old style Java implementation

If a Java archive (`.jar`) file with an old style java implementation is used, the Java support must not be activated in the app.ducx project.

For all defined Java implementation neither the package string is verified nor the prefix is generated For further information on how to attach a trigger action to a trigger event in a property definition, please refer to chapter 8.4.1 “Defining a use case to be implemented in Java”.

If the project is loaded into a Fabasoft domain, the Java archive (`.jar`) file has to be copied manually to the web server.

8.4.13 Working with type definition of a customization point

The type of a customization point is accessible with the class `TypeCustomizationPointDef`.

Example

```
public List<ActionParameterList> GetNameBuildInParameters() throws CooException {
  //Get the type of the customization point
  TypeCustomizationPointDef cptdef = FSCCONFIG 1 1001.getCPT NameBuild();
  //Return customization point parameters
  List<ActionParameterList> parameters = tcpt.COOSYSTEM_1_1_typecpparameters;
  ...
  return inparameters;
}
```

The customizations in a customization point can only be accessed through a use case implementation with Fabasoft Expression.

8.5 Overriding an existing use case implementation

Syntax

```
override usecase {
  variant objectclass {
    impl = ...
  }
}
```

You can override an existing use case implementation for the object classes belonging to your software component.

When overriding an existing use case implementation, the `override` keyword must precede the reference of the use case that you want to override, followed by curly braces.

Cloud profile note: Overriding an external use case from a non-friend component for an external class from a non-friend component is forbidden.

In the following example, a custom implementation is defined for `COOSYSTEM@1.1:AttrContentSet` in object class `APPDUCXSAMPLE@200.200:Invoice`.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  override AttrContentSet {
    variant Invoice {
      impl = expression {
        // Call super method
        cooobj.CallMethod(cootx, coometh);
        if (attrdef == #content) {
          // Initialize invoice approval process
          cooobj.COOWF@1.1:InitializeWorkFlow([#APPDUCXSAMPLE@200.200:
            ProcInvoiceApproval]);
        }
      }
    }
  }
}
```

8.6 Use case wrappers

Use case wrappers allow defining reusable building blocks with a standard implementation and polymorphism to override the default behavior.

A use case wrapper may define a prototype, a virtual application prototype, a method definition or a virtual application.

- `COOSYSTEM@1.1:ucwprototype`
- `FSCVAPP@1.1001:ucwprototype`
- `COOSYSTEM@1.1:ucwmethdefinition`
- `FSCVAPP@1.1001:ucwapplication`

Cloud profile note: The definition of use case wrappers is not allowed.

Example

app.ducx Use Case Language

```
/**
```

```
 * Default implementation for signature wrappers. If no selection is supplied,
 * it is applied to "sys object". If "signtype" is not available for any
 * reason, an exception is thrown.
 */
usecase SignObjects(parameters as FSCVAPP@1.1001:MenuPrototype) {
  variant Object {
    impl = application {
      expression {
        if (!sys selobjects) {
          sys selobjects = sys object;
        }
        Action sys_action;
        SignatureType signtype = sys_action.signtype;
        if (!signtype) {
          throw #SIGNERR IllegalType;
        }
        ->SignSelectedObjectsApp(sys_object, sys_action, sys_view,
              sys_selobjects, sys_selindices, sys_dynkey, signtype, null);
      }
    }
  }
}
```

app.ducx Object Model Language

```
/**
 * Defines a signature wrapper with a default virtual application
 * implementation ("SignObjects").
 */
class<UseCaseWrapper> SignatureWrapper : UseCase {
  ucwapplication = ObjectSignObjects;
  SignatureType signtype not null;
}
```

app.ducx Use Case Language

```
/**
 * "SignWithMenuInitial" creates a menu use case; it is implemented as
 * virtual application defined in "SignatureWrapper"
 */
menu usecase<SignatureWrapper> SignWithMenuInitial {
  signtype = SIGN_INITIAL;
}
```

8.7 Use case wrappers (old style)

Syntax

```
usecase {
  prewrappers = {
    prewrapper,
    ...
  }
  postwrappers = {
    postwrapper,
    ...
  }
}
```

Fabasoft app.ducx allows you to add a use case wrapper to an existing use case that is invoked whenever the wrapped use case is executed.

Note: Wrappers are only allowed if the software component has a 1.* domain ID (e.g. `DUCXSAMP@1.1001`).

Two types of use case wrappers are supported:

- The `prewrappers` keyword is used to assign one or more prewrappers to an existing use case. Multiple entries must be separated by colons. A prewrapper is invoked before the wrapped use case is executed.
- The `postwrappers` keyword is used to assign one or more postwrappers to an existing use case. Multiple entries must be separated by colons. A postwrapper is invoked after the wrapped use case has been executed successfully.

Note: A use case wrapper must be assigned the same prototype or parameter list as the use case to be wrapped by the use case wrapper.

Cloud profile note: The definition of use case wrappers is not allowed.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  VerifyContent(parameters as AttrSetPrototype) {
    variant Object {
      impl = java:APPDUCXSAMPLE.Object.VerifyContent;
    }
  }
  override AttrContentSet {
    postwrappers = {
      VerifyContent
    }
  }
}
```

8.8 Implementing triggers

There are two types of triggers in Fabasoft Folio:

- object-level triggers fired for specific events involving objects
- property-level triggers fired for specific events involving properties

Triggers are executed in the background and therefore cannot involve user interaction. Furthermore, please note that triggers must not be implemented as virtual applications.

Note: Triggers should be defined as actions. Omit the `usecase` keyword when defining new triggers using the app.ducx use case language.

8.8.1 Object-level triggers

Object-level triggers are defined in object class `COOSYSTEM@1.1:Object` and are invoked automatically by Fabasoft Folio.

You can override object-level triggers for your object classes to change or add to their default behavior.

When overriding an object-level trigger, the fully qualified reference of the trigger action must be denoted, followed by curly braces.

The implementation of an overridden object-level trigger should usually call the super method. In app.ducx expression language, this can be accomplished by the following statement:

```
cooobj.CallMethod(cootx, coometh);
```

8.8.1.1 Constructor trigger

The object constructor trigger `COOSYSTEM@1.1:ObjectConstructor` is fired when a new instance of an object class is created. The trigger is invoked on the new instance.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  override ObjectConstructor {
    variant Order {
      impl = expression {
        // Call super method
        cooobj.CallMethod(cootx, coometh);
        // Add order to the list of orders stored in the current user's
        // desk object (expecting that APPDUCXSAMPLE@200.200:userorders has
        // been added as a property of the desk object class)
        root = cooroot;
        if (root.HasAttribute(cootx, #userorders)) {
          cooobj.COODESK@1.1:ShareObject(null, null, #userorders, root);
        }
      }
    }
  }
}
```

The next example illustrates a Java method implementation of the same functionality as shown in the previous example.

Example

app.ducx Use Case Language

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  override ObjectConstructor {
    variant Order {
      impl = java:APPDUCXSAMPLE.Order.OrderConstructor;
    }
  }
}
```

Java Implementation

```
@DUCXImplementation("COOSYSTEM@1.1:ObjectConstructor")
public void OrderConstructor(final Object sourceobj) throws CooException {
  // Call super method
  super.COOSYSTEM_1_1_ObjectConstructor(sourceobj);
  // Add order to the list of orders stored in the current user's
  // desk object (expecting that APPDUCXSAMPLE@200.200:userorders has
  // been added as a property of the desk object class)
  RootObject root = RootObject.from(coort.GetCurrentUserRoot());
  if (root.HasAttribute(cootx, APPDUCXSAMPLE_200_300.getProperty_userorders)) {
    this.COODESK_1_1_ShareObject(null, null, APPDUCXSAMPLE_200_300.
      getProperty_userorders, root);
  }
}
```

8.8.1.2 Prepare commit trigger

The prepare commit trigger `COOSYSTEM@1.1:ObjectPrepareCommit` is fired before any set triggers are invoked. The trigger is invoked on the object that is to be changed.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  override ObjectPrepareCommit {
    variant Order {
      impl = expression {
        // Generate the order's name based on the name build configuration
        cooobj.FSCCONFIG@1.1001:ObjectPrepareCommitNameBuild();
      }
    }
  }
}
```

The next example illustrates a Java method implementation of the same functionality as shown in the previous example.

Example

app.ducx Use Case Language

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  override ObjectPrepareCommit {
    variant Order {
      impl = java:APPDUCXSAMPLE.Order.OrderPrepareCommit;
    }
  }
}
```

Java Implementation

```
@DUCXImplementation("COOSYSTEM@1.1:ObjectPrepareCommit")
public void OrderPrepareCommit(final Boolean internalchange)
  throws CooException {
  // Generate the order's name based on the name build configuration
  this.FSCCONFIG_1_1001_ObjectPrepareCommitNameBuild();
}
```

8.8.1.3 Finalize commit trigger

The finalize commit trigger `COOSYSTEM@1.1:ObjectFinalizeCommit` is fired after the set triggers have been invoked, but before any changes are committed to an object. The trigger is invoked on the object that is to be changed.

Example

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  override ObjectFinalizeCommit {
    variant Order {
      impl = expression {
        // Call super method
        cooobj.CallMethod(cootx, coometh);
        // Enforce that users also have to change the list of order
        // positions when changing the vendor
        if (cootx.HasAttributeChanged(cooobj, #ordervendor)
            && !cootx.HasAttributeChanged(cooobj, #orderpositions) {
          throw coort.SetError(#InvalidChange, null);
        }
      }
    }
  }
```

```
}
```

The next example illustrates a Java method implementation of the same functionality as shown in the previous example.

Example

app.ducx Use Case Language

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  override ObjectFinalizeCommit {
    variant Order {
      impl = java:APPDUCXSAMPLE.Order.OrderFinalizeCommit;
    }
  }
}
```

Java Implementation

```
@DUCXImplementation("COOSYSTEM@1.1:ObjectFinalizeCommit")
public void OrderFinalizeCommit(final List<Object> attrlist)
  throws CooException {
  // Call super method
  super.COOSYSTEM_1_1_ObjectFinalizeCommit(attrlist);
  // Enforce that users also have to change the list of order
  // positions when changing the vendor
  if (cootx.HasAttributeChanged(this, APPDUCXSAMPLE_200_300.
    getProperty_ordervendor()) && !cootx.HasAttributeChanged(this,
    APPDUCXSAMPLE 200 300.getProperty orderpositions)) {
    coort.SetError(APPDUCXSAMPLE_200_300.getErrorMesssage_InvalidChange());
  }
}
```

8.8.2 Property-level triggers

Property-level triggers are fired only when they are explicitly attached to a property for a predefined trigger event.

For defining a property-level trigger, you have to follow these steps:

- Create and implement a use case for the trigger
- Attach the trigger action to the trigger event in the property definition

For further information on how to attach a trigger action to a trigger event in a property definition, please refer to chapter 5.3.3 “Assigning triggers to a property”.

8.8.2.1 Constructor trigger

When a new instance of an object class is created, the constructor triggers of all properties are fired – provided that a constructor trigger has been defined for the particular property.

For a constructor trigger action, you have to assign the `COOSYSTEM@1.1:AttrConstructorPrototype`. The initialization value that is to be assigned to the property must be returned in the second parameter.

Example

app.ducx Object Model Language

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
```

```
  class Order : CompoundObject {
    OrderState orderstate readonly(ui) {
      ctor = OrderStateCtor;
    }
  }
}
```

app.ducx Use Case Language

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  OrderStateCtor(parameters as AttrConstructorPrototype) {
    variant Order {
      impl = expression {
        // Initialize order state as pending
        value = "OS_PENDING";
      }
    }
  }
}
```

The next example illustrates a Java method implementation of the same functionality as shown in the previous example.

Example

app.ducx Object Model Language

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class Order : CompoundObject {
    OrderState orderstate readonly(ui) {
      ctor = OrderStateCtor;
    }
  }
}
```

app.ducx Use Case Language

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  OrderStateCtor(parameters as AttrConstructorPrototype) {
    variant Order {
      impl = java:APPDUCXSAMPLE.Order.OrderStateCtor;
    }
  }
}
```

Java Implementation

```
@DUCXImplementation("APPDUCXSAMPLE@200.200:OrderStateCtor")
public OrderStateCtorResult OrderStateCtor(final Object attrdef)
  throws CooException {
  // Initialize order state as pending
  OrderStateCtorResult result = new OrderStateCtorResult();
  result.value = new ArrayList<java.lang.Object>();
  result.value.add(OrderState.OS_PENDING);
  return result;
}
```

8.8.2.2 Get value trigger

The get value trigger is fired, after the property value is read from the database.

For a get value trigger action, you have to assign the `COOSYSTEM@1.1:AttrGetPrototype`. The value read from the database is passed to the trigger in the second parameter. The implementation can then modify the value, and return the modified value – also in the second parameter.

Instead of defining a constraint, the example shown can also be implemented using a get value trigger as illustrated in the following example.

Example

app.ducx Object Model Language

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class Order : CompoundObject {
    currency ordertotal readonly volatile {
      get = GetOrderTotal;
    }
  }
}
```

app.ducx Use Case Language

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  GetOrderTotal(parameters as AttrGetPrototype) {
    variant Order {
      impl = expression {
        for (OrderPosition position : cooobj.orderpositions) {
          Product product = position.product;
          if (product != null) {
            currency total += product.unitprice * position.quantity;
          }
        }
        total;
      }
    }
  }
}
```

The next example illustrates a Java method implementation of the same functionality as shown in the previous example.

Example

app.ducx Object Model Language

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class Order : CompoundObject {
    currency ordertotal readonly volatile {
      get = GetOrderTotal;
    }
  }
}
```

app.ducx Use Case Language

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  GetOrderTotal(parameters as AttrGetPrototype) {
    variant Order {
      impl = java:APPDUCXSAMPLE.Order.GetOrderTotal;
    }
  }
```

```
}
```

Java Implementation

```
@DUCXImplementation("APPDUCXSAMPLE@200.200:GetOrderTotal")
public GetOrderTotalResult GetOrderTotal(final Object attrdef,
  final List<java.lang.Object> value) throws CooException {
  GetOrderTotalResult result = new GetOrderTotalResult();
  Currency ordertotal = Currency.create();
  long amount = 0;
  List<OrderPosition> positions = this.APPDUCXSAMPLE_200_300_orderpositions;
  for (OrderPosition position : positions) {
    Long quantity = position.APPDUCXSAMPLE_200_300_quantity;
    Product product = position.APPDUCXSAMPLE 200 300 product;
    Currency unitprice = product.APPDUCXSAMPLE 200 300 unitprice;
    long price = Long.parseLong(unitprice.COOSYSTEM_1_1_currvalue);
    amount += quantity * price;
  }
  ordertotal.COOSYSTEM_1_1_currsymbol = CurrencySymbol.USD;
  ordertotal.COOSYSTEM_1_1_currvalue = Long.toString(amount);
  result.value = new ArrayList<java.lang.Object>();
  result.value.add(ordertotal);
  return result;
}
```

8.8.2.3 Set value trigger

The set value trigger is fired, before the property value is written to the database.

For a set value trigger action, you have to assign the `COOSYSTEM@1.1:AttrSetPrototype`. The current value of the property is passed to the trigger in the second parameter. The value contained in the property before it was changed is made available in the third parameter. If the value to be written to the database is changed by the implementation of the trigger, it must be returned in the second parameter.

Example

app.ducx Object Model Language

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class<ContentObjectClass> : ContentObject {
    date invoicepaymentdate {
      set = SetInvoiceDate;
    }
  }
}
```

app.ducx Use Case Language

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  SetInvoiceDate(parameters as AttrSetPrototype) {
    variant Invoice {
      impl = expression {
        if (value != null && value < coonow) {
          cooobj.ObjectLock(true, true);
          cooobj.invoicestate = "IS_PAID";
        }
      }
    }
  }
}
```

The next example illustrates a Java method implementation of the same functionality as shown in the previous example.

Example

app.ducx Object Model Language

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  class<ContentObjectClass> : ContentObject {
    date invoicepaymentdate {
      set = SetInvoiceDate;
    }
  }
}
```

app.ducx Use Case Language

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  SetInvoiceDate(parameters as AttrSetPrototype) {
    variant Invoice {
      impl = java:APPDUCXSAMPLE.Invoice.SetInvoiceDate;
    }
  }
}
```

Java Implementation

```
@DUCXImplementation("APPDUCXSAMPLE@200.200:SetInvoiceDate")
public SetInvoiceDateResult SetInvoiceDate(final Object attrdef,
  final List<java.lang.Object> value, final List<java.lang.Object>
  oldvalue) throws CooException {
  SetInvoiceDateResult result = new SetInvoiceDateResult();
  Date invoicedate = (Date) value.get(0);
  if (invoicedate != null && invoicedate.before(new Date())) {
    this.COOSYSTEM_1_1_ObjectLock(true, true, null, null);
    this.APPDUCXSAMPLE_200_300_invoicestate = InvoiceState.IS_PAID;
  }
  return result;
}
```

9 app.ducx Organizational Structure Language

The abstract elements of the organizational structure required for your solution are defined using the app.ducx organizational structure language.

Fabasoft Folio allows you to represent an organizational structure using abstract and concrete structure elements. These organizational structure elements can be used for assigning access rights and for defining actors in a workflow.

The following abstract elements are defined using the app.ducx organizational structure language and can be used to model an organizational hierarchy independently of users and groups:

- organizational units define abstract areas of an organization (e.g. "manufacturing", "sales", "department")
- positions are used to split up organizational units into functional tasks and areas of responsibility (e.g. "production manager" in organizational unit "manufacturing", "customer representative" in organizational unit "sales").

The benefit of the abstract elements is that you can model the organizational structure without taking into account actual users and groups. The concrete elements of the organizational structure – users

and groups – can be linked to the abstract elements when your software component is deployed to the customer.

In a customer installation, groups can be linked to abstract organizational units, and users can be assigned to groups. A user can be member of one or more groups. Moreover, roles can be assigned to each user. A role is defined as the position that a user can occupy within a given group.

For instance, assume that Jane Bauer is the manager of the Sales business unit. This can be modeled in Fabasoft Folio by assigning a role to user Jane Bauer that is comprised of position “Manager” and group “Sales”. Furthermore, the “Sales” group must be assigned to organizational unit “Business Unit”.

Note: Users and groups are always created in the customers’ Fabasoft Folio Domains whereas the abstract elements, positions and organizational units, are defined using the app.ducx organizational structure language, and shipped with your software component.

An organizational structure model block consists of import declarations and organizational structure elements. The `orgmodel` keyword denotes an organizational structure model block. It must be followed by the reference of your software component and curly braces.

Organizational structure model blocks can only be contained in files with a `.ducx-os` extension.

Cloud profile note: Organizational structure models are not supported.

Syntax

```
orgmodel softwarecomponent
{
  // Import declarations
  import softwarecomponent;
  // Organizational structure model elements (positions, organizational
  // units)
  ...
}
```

9.1 Defining a position

Syntax

```
position reference;
```

The `position` keyword is used to define a position. It must be followed by a reference and a semicolon.

Example

```
orgmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  position Clerk;
  position DeptManager;
  position DeptSecretary;
}
```

9.2 Defining an organizational unit

Syntax

```
orgunit reference {
  positions = {
```

```
    position,
    ...
  }
}
```

The `orgunit` keyword is used to define an organizational unit. It must be followed by a reference and curly braces.

The `positions` keyword allows you to assign positions to an organizational unit. Multiple entries must be separated by commas.

Example

```
orgmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  orgunit OrderProcessing {
    positions = {
      Clerk,
      DeptManager,
      DeptSecretary
    }
  }
}
```

9.3 Extending an organizational unit

Syntax

```
extend orgunit reference {
  positions = {
    position,
    ...
  }
}
```

With the `extend orgunit` keywords, you can add positions to an organizational unit that is part of another software component.

Example

```
orgmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCFOLIO@1.1001;
  extend orgunit ManagementOU {
    positions = {
      DeptSecretary
    }
  }
}
```

9.4 Defining an access type

Syntax

```
acctype reference {
  finalform = booleanvalue;
  sequence = sequencenumber;
  symbol = symbol;
}
```

Access types are used to secure read, change and execute access to properties or use cases. If a property or use case is protected by a custom access type, the user must be granted this access type by the object's ACL in order to access the property or to invoke the use case.

Note: Software component `COOSYSTEM@1.1` already provides a set of access types that can be reused for protecting your properties and use cases.

The `acctype` keyword is used to define an access type. It must be followed by a reference and curly braces.

Within an `acctype` block, the `finalform` keyword is used to specify whether or not a user is able to access completed objects with this access type. If `finalform` is set to `false`, a user cannot access objects in final form with this access type.

The `sequence` keyword is used for defining the sequence number of the access type within the software component. If multiple access types are defined by a software component, the sequence number can be used for determining the order in which they are displayed in the ACL editor.

The `symbol` keyword is used for assigning a symbol to an access type. The symbol assigned to an access type is displayed in the ACL editor.

Example

```
orgmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  acctype AccTypeApproveOrder {
    finalform = true;
    sequence = 1;
    symbol = SymbolApprove;
  }
}
```

10 app.ducx Business Process Language

The app.ducx business process language allows you to define the process model for your software component. The process model is comprised of one or more template processes defining an ordered list of sequential or parallel activities and process control elements such as conditions or loops. Each activity definition consists of a list of steps that need to be carried out by a user responsible for completing a particular activity.

A process model block consists of import declarations and `process` blocks. The `processes` keyword denotes the process model block. It must be followed by the reference of your software component and curly braces.

Process model blocks can only be contained in files with a `.ducx-bp` extension.

Syntax

```
processes softwarecomponent
{
  // Import declarations
  import softwarecomponent;
  // Process blocks
  process reference {
    symbol = symbol;
    allowed {
      objectclass,
      ...
    }
    denied {
      objectclass,
```

```
      ...
    }
    ...
  }
}
```

10.1 Defining a process

A process definition is comprised of a sequence of serialized or parallel activities and control elements. Using such a process definition, a process instance can be instantiated by a user. A process instance is usually attached to exactly one business object.

The `process` keyword must be used to define a process. It must be followed by the reference and curly braces.

A `process` block may contain referenced processes (sub-processes), referenced activities, activity definitions and control elements. The following elements are supported within a `process` block:

- activities defined using the `activity` keyword or referenced activities
- referenced processes (sub-processes)
- conditions defined using the `if` keyword
- case statements defined using the `switch` keyword
- loops defined using the `repeat` keyword
- sub processes also defined using the `activity` keyword
- parallel blocks defined using the `parallel` keyword

In addition to activity definitions and control elements a `process` block also supports the following elements:

- The `symbol` keyword can be used to assign a custom symbol to a process definition.
- The `allowed` keyword can be used to define a list of allowed object classes. If you define a list of allowed object classes, the process definition may only be used for initializing processes on instances of the object classes in the `allowed` block.
- The `denied` keyword can be used to define a list of excluded object classes. If you define a list of excluded object classes, the process definition may not be used for initializing processes on instances of the object classes in the `denied` block.
- With `common = true` the process can be marked as commonly useable.

Cloud profile note: Conditions, case statements, loops are not supported.

10.2 Defining an activity

The `activity` keyword is used to define an activity. It can be nested directly within the `processes` block to define an activity used for ad hoc prescriptions or to define an activity that should be referenced in several `process` blocks. Within a `process` block you can define an activity that is part of a process.

When defining an activity within the `processes` block, the `activity` keyword must be followed by the reference and curly braces.

The `symbol` keyword can be used to assign a custom symbol to an activity. By adding `common = true` activities can be marked as commonly useable.

Syntax

```
activity {
```

```
  actor {
    meta = abstractactor;
    pos = position;
    orgunit = organizationalunit;
  }
  step stepidentifier stepmodifiers {
    precond = expression {
      ...
    }
    execute = ...
  }
}
```

The simplest process definitions are comprised of a sequence of `activity` blocks that are processed one after the other.

Example

```
processes APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COOWF@1.1;
  // Definition of an activity for ad hoc prescriptions
  activity AdHocProcessing {
    ...
  }
  // Definition of an activity that should be used within several processes
  activity CommonActivity {
    ...
  }
  // Definition of a process comprised of three sequential activities
  process OrderWF {
    // Definition of the first activity of the process
    activity {
      ...
    }
    // Definition of the second activity of the process
    activity {
      ...
    }
    // Definition of the third activity of the process
    activity {
      ...
    }
    // Use of CommonActivity as the forth activity of the process
    CommonActivity;
  }
}
```

10.2.1 Defining the actor

Each activity must have an `actor` block defining the actor responsible for the completion of the activity.

The actor can either be an abstract entity – such as the process initiator or the process owner – or what is referred to as a *role* in Fabasoft terminology, which is combination of position and organizational unit.

Note: An actor can also be defined in referenced activities. In this case the predefined actor of the referenced activity definition is overridden.

The following keywords can be used to define an actor:

- `meta` for referencing an abstract entity as listed in the next table
- `pos` for referencing a position

- `orgunit` for referencing an organizational unit

Abstract actor	Description
`WFMP_INITIATOR`	Process initiator
`WFMP_RESPONSIBLE`	Actor responsible for process
`WFMP_PROCOWNER`	Process owner
`WFMP_PROCGROUP`	Process group
`WFMP_OBJOWNER`	Object owner
`WFMP_OBJGROUP`	Object group
`WFMP_CURRENTUSER`	Current user
`WFMP_PARTICIPANT`	Current actor

Table 32: Abstract actors

Note: In addition to the abstract actors listed in the previous table, software component developers can also define custom abstract actors.

Cloud profile note: The definition of position and organizational unit is not allowed.

10.2.2 Defining the steps to be executed

An activity consists of a list of steps that can either be optional or required. Required steps must be executed before the actor is allowed to complete the activity.

For each step, a `step` block must be defined within the `activity` block. The `step` keyword can be followed by an optional identifier for the step, and curly braces.

The `execute` keyword allows you to define the “implementation” for a step. Put simply, `execute` is used to define which action is carried out when the step is executed by the user.

A step can be implemented inline using app.ducx expression language. Alternatively, you can also reference a use case or a virtual application.

Note: The use case or virtual application is invoked on the business object attached to the process. If the step is implemented in app.ducx expression language, the business object can be accessed using the `cooobj` variable.

If you do not provide an implementation for a step by omitting the `execute` keyword, a dialog box is presented to the user upon execution asking whether the step has been carried out successfully.

10.2.2.1 Transaction variables holding the context

When executing a step of an activity, the Fabasoft Folio Workflow Engine makes available several transaction variables holding context information (see chapter 8.4.10.2 “Working with transaction variables”).

The following example demonstrates how to access the transaction variables provided by the Fabasoft Folio Workflow Engine from a virtual application that is used to implement a step of an activity.

Example

app.ducx Use Case Language

```
usecases APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COOWF@1.1;
  usecase EditOrderWF() {
    variant Order {
      impl = application {
        expression {
          Object obj = #TV.WFVAR_THIS;
          ProcessInstance process = #TV.WFVAR_PROCESS;
          ActivityInstance activity = #TV.WFVAR_ACTIVITY;
          integer workitemidx = #TV.WFVAR_WORKITEM;
          process->COODESK@1.1:OpenObject();
        }
      }
    }
  }
}
```

app.ducx Business Process Language

```
processes APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COOWF@1.1;
  process OrderWF {
    symbol = SymbolOrderWF;
    allowed {
      Order
    }
    activity {
      actor {
        meta = WFMP_INITIATOR;
      }
      step AssignOrder {
        execute = EditOrderWF;
      }
    }
  }
}
```

10.2.2.2 Using an expression to implement a step

When implementing a step using app.ducx expression language, the local scope `this` is populated with a dictionary. The entries are listed in. Hence, you do not need to read the transaction variables listed in the next table. Instead, you can also obtain the values from the dictionary.

Key	Description
`object`	Business object attached to the current process
`process`	Process instance
`activity`	Activity instance
`workitem`	Zero-based index of the step executed

Table 33: Local scope for steps implemented as expression

10.2.2.3 Defining a precondition for a step

The `precond` keyword is used to define a precondition that must evaluate to `true` for the step to be enabled in the user interface. If you do not define a precondition for a step, it is enabled by default.

The precondition must be implemented using app.ducx expression language. The previous table lists the entries available in the local scope `this` when the expression is evaluated. If the expression returns `true`, the step is enabled in the user interface, otherwise it is disabled.

In the following example, the second step of the activity is enabled only after the first step has been executed successfully. This is achieved by defining a precondition expression that checks whether the completion date property (`COOWF@1.1:wfwcompletedat`) has been populated for the first step (`COOWF@1.1:actinstwork[0]`) of the activity.

Example

```
processes APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COOWF@1.1;
  process OrderWF {
    // Activity for checking the order
    activity {
      actor {
        meta = WFMP_INITIATOR;
      }
      step ViewOrder {
        execute = ViewOrder;
      }
      step ReleaseOrder {
        precond = expression {
          activity.actinstwork[0].wfwcompletedat != null
        }
        execute = ReleaseOrder;
      }
    }
  }
}
```

10.2.2.4 Defining additional settings for a step

For each step of an activity, you can define whether it should be required, repeatable and whether the current activity should be completed automatically after the step has been executed successfully.

These settings are referred to as step modifier suffixes, which are denoted following the step identifier or the `step` keyword if the step identifier was omitted. Multiple step modifier suffixes must be separated by whitespaces.

10.2.2.4.1 Defining a required step

For defining a required step, the `required` keyword must be appended as a step modifier suffix.

All required steps must be executed before an activity can be completed by the user.

10.2.2.4.2 Defining a repeatable step

By default, steps can only be executed once. Once executed successfully, a step cannot be executed again except if the `multiple` keyword is appended as a step modifier suffix.

10.2.2.4.3 Defining an auto-completing step

If the `leave` step modifier suffix is added to a step, the activity is completed automatically after the concerned step has been executed successfully.

If an activity does not have any auto-completing steps, it must be completed manually for the workflow to proceed with the next activity in the process.

Note: An activity can only be completed after all required steps have been executed by the user even if the `leave` step modifier suffix was added for the executed step.

Example

```
processes APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COOWF@1.1;
  process OrderWF {
    // Activity for checking the order
    activity {
      actor {
        meta = WFMP_INITIATOR;
      }
      step ViewOrder multiple {
        execute = ViewOrder;
      }
      step ReleaseOrder required leave {
        execute = ReleaseOrder;
      }
    }
    // Activity for shipping the order
    activity {
      actor {
        pos = Clerk;
        orgunit = OrderProcessing;
      }
      step ViewOrder multiple {
        execute = ViewOrder;
      }
      step MarkOrderAsShipped required leave {
        execute = MarkOrderAsShipped;
      }
    }
    // Activity for billing the order
    activity {
      actor {
        pos = Clerk;
        orgunit = Accounting;
      }
      step ViewOrder multiple {
        execute = ViewOrder;
      }
      step SendInvoice required leave {
        execute = SendInvoice;
      }
    }
  }
}
```

10.2.3 Extending an activity

The `extend` keyword followed by the keyword `activity` allows you to extend own activities and activities of other software components.

Syntax

```
extend activity reference {
  ...
}
```

Cloud profile note: The extension of activities that belong to another non-friend software component is not allowed.

In the following example an activity is extended by a step.

Example

```
extend activity CreateNotification {
  step FinalizeDocument {
    precond = expression {
      activity.actinstwork[2].wfwcompletedat != null;
    }
    execute = expression {
      object.casestate = 4;
    }
  }
}
```

10.3 Defining conditions, case statements and loops

A process can include several types of control elements that allow you to define which path should be taken in a process depending on the result of a predefined expression.

10.3.1 Defining a condition

Syntax

```
if (condition) {
  // Activity definitions or control elements
  ...
}
else {
  // Activity definitions or control elements
  ...
}
```

The `if` and `else` keywords denote a condition in a process. A conditional statement embedded into a process allows you to control which path should be taken in a process depending on whether the expression defined in the condition evaluates to `true` or `false`.

If the condition evaluates to `true`, the block following the `if` statement is executed. Otherwise, the execution continues in the `else` block. The `else` block, however, is optional.

When the condition is evaluated, a dictionary is made available in the local scope `this` that contains the entries listed in chapter 10.2.2.2 “Using an expression to implement a step”– except for `workitem`, which is not available in this context.

Example

```
processes APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COOWF@1.1;
  process OrderWF {
    // Activity for approving an order
    activity {
      actor {
        pos = DeptManager;
        orgunit = OrderProcessing;
      }
      step ViewOrder multiple {
        execute = ViewOrder;
      }
      step ApproveOrder required leave {
        execute = expression {
          object.ObjectLock(true, true);
          object.orderstate = OrderState(OS_APPROVED);
        }
```

```
      }
      step DenyApproval required leave {
        execute = expression {
          object.ObjectLock(true, true);
          object.orderstate = OrderState(OS_DISCARDED);
        }
      }
    }
    if (object.orderstate == OrderState(OS_APPROVED)) {
      // Activity for shipping the order
      activity {
        ...
      }
      // Activity for billing the order
      activity {
        ...
      }
    }
    else {
      // Activity for handling discarded orders
      activity {
        ...
      }
    }
  }
}
```

10.3.2 Defining a case statement

Syntax

```
switch (enumerationproperty) {
  case enumerationitem:
    // Activity definitions or control elements
    ...
  default:
    // Activity definitions or control elements
    ...
}
```

A case statement allows you to define different process paths depending on the value of an enumeration property of the business object the process is attached to.

The `switch` keyword is used to define a case statement. It must be followed by parentheses and curly braces, and the reference of the enumeration property to be evaluated must be enclosed in the parentheses.

The `switch` block may contain `case` sections for each enumeration item. The `case` keyword must be followed by the reference of the enumeration item and a colon. You can also define a single `case` section for multiple enumeration items by separating the enumeration items by a comma. The `default` keyword can be used to define a default section applying to all enumeration values that are not handled in an appropriate `case` section.

Example

```
processes APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COOWF@1.1;
  process CustomerWF {
    switch (customertype) {
    case CT_ENDUSER:
      activity {
        step AssignOrderToEndUser {
          ...
```

```
        }
      }
    case CT_RETAILER, CT_WHOLESALER:
      activity {
        step AssignOrderToReseller {
          ...
        }
      }
    default:
      activity {
        step AssignOrder {
          ...
        }
      }
    }
  }
}
```

10.3.3 Defining a loop

Syntax

```
repeat {
  // Activity definitions or control elements
  ...
} until (terminationcondition);
```

The `repeat` and `until` keywords can be used to define a loop in a process. The loop is executed until the expression defined in parentheses after the `until` keyword – referred to as termination condition – evaluates to `false`.

When the termination condition is evaluated, a dictionary is made available in the local scope `this` that contains the entries listed in chapter 10.2.2.2 “Using an expression to implement a step”– except for `workitem`, which is not available in this context.

Example

```
processes APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COOWF@1.1;
  process CustomerWF {
    repeat {
      // Activities and control elements for processing an order
      activity {
        ...
      }
      activity {
        ...
      }
    } until (object.orderstate == OrderState(OS_COMPLETED));
  }
}
```

10.3.4 Defining a gateway

Syntax

```
gateway {
  type = gatewaytype;
  // Path definitions
  path (pathcondition) {
```

```
    reference = "reference1";
    // Activity definitions or control elements
    end;
  }
  path (pathcondition) {
    reference = "reference2";
    // Activity definitions or control elements
    end;
  }
  default {
   reference = "reference3";
    // Activity definitions or control elements
  }
}
```

The `gateway` keyword can be used to define exclusive and inclusive gateways in a process. If the type equals `GWT_EXCLUSIVE`, the conditions of the paths are evaluated and only the first path, where the condition evaluated to true, will be executed. Choosing the type `GWT_INCLUSIVE` leads to the result that all paths with a condition evaluated to true will be executed.

Example

```
processes APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COOWF@1.1;
  process CustomerWF {
    gateway {
      type = GWT_EXCLUSIVE;
      path (cooobj.objname contains "Default") {
        reference = "reference1";
        sequence {
          activity {
            ...
          }
        }
        end;
      }
      default {
        reference = "reference1";
        sequence {
          activity {
            ...
          }
        }
      }
    }
  }
}
```

Paths of gateways can lead to the explicit termination of the current process. This can be denoted by the keywords “end”, “terminate” or “error”. “end” ends the current process, “terminate” terminates the execution of the current process and “error” indicates that the process needs to be terminated due to an eror.

10.4 Defining parallel activities

The Fabasoft Folio Workflow Engine allows you to define processes that are comprised of parallel activities and control elements.

10.4.1 Defining a block of parallel activities

Syntax

```
parallel {
  // Activity definitions or control elements
  ...
}
```

The `parallel` keyword is used to define blocks of parallel activities. `parallel` blocks can be nested within a `process` block, within control element blocks or within other `parallel` blocks.

Example

```
processes APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COOWF@1.1;
  process ApproveOrderWF {
    parallel {
      activity {
        actor {
          pos = Clerk;
          orgunit = OrderProcessing;
        }
        step ApproveOrder required leave {
          execute = ApproveOrder;
        }
        step DenyApproval required leave {
          execute = DenyOrderApproval;
        }
      }
      activity {
        actor {
          pos = DeptManager;
          orgunit = OrderProcessing;
        }
        step ApproveOrder required leave {
          execute = ApproveOrder;
        }
        step DenyApproval required leave {
          execute = DenyOrderApproval;
        }
      }
    }
  }
}
```

A process can be split at any time using a `parallel` block. The default behavior when joining parallel process paths is that all parallel activities must be completed before the workflow continues with the next non-parallel activity in the process.

10.4.2 Defining activity sequences within a parallel block

Syntax

```
parallel {
  // Activity definitions or control elements
   ...
  sequence {
    // Activity definitions or control elements
    ...
  }
  // Activity definitions or control elements
```

```
  ...
}
```

The `sequence` keyword is used to define a sequence of activities within a `parallel` block. It must be followed by a block defining the sequence of activity definitions and control elements within a path of a `parallel` block. A `sequence` block can also contain another `parallel` block for splitting the current process path.

Example

```
processes APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COOWF@1.1;
  process ApproveOrderWF {
    parallel {
      activity {
        actor {
          pos = Clerk;
          orgunit = OrderProcessing;
        }
        step ApproveOrder required leave {
          execute = ApproveOrder;
        }
        step DenyApproval required leave {
          execute = DenyOrderApproval;
        }
      }
      sequence {
        activity {
          actor {
            pos = DeptSecretary;
            orgunit = OrderProcessing;
          }
          step ApproveOrder required leave {
            execute = ApproveOrder;
          }
          step DenyApproval required leave {
            execute = DenyOrderApproval;
          }
        }
        if (object.preapproved) {
          activity {
            actor {
              pos = DeptManager;
              orgunit = OrderProcessing;
            }
            step ApproveOrder required leave {
              execute = ApproveOrder;
            }
            step DenyApproval required leave {
              execute = DenyOrderApproval;
            }
          }
        }
      }
    }
  }
}
```

10.5 Defining sub processes

The Fabasoft Folio Workflow Engine allows you to invoke a sub process from within a process. An activity is used as a container to embed a sub process within a process. The sub process is expanded on demand when the workflow reaches the activity acting as a container for the sub process.

A sub process is defined just like an ordinary process using the `process` keyword.

Syntax

```
activity {
  subproc = subprocess;
}
```

The `activity` keyword is used to embed a sub process within a process, but instead of defining actor and steps for the activity you just need to reference the sub process using the `subproc` keyword.

Example

```
processes APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import COOWF@1.1;
  // Definition of sub process APPDUCXSAMPLE@200.200:DeptMgrApprovalWF
  process DeptMgrApprovalWF {
    activity {
      actor {
        pos = DeptSecretary;
        orgunit = OrderProcessing;
      }
      step ApproveOrder required leave {
        execute = ApproveOrder;
        }
        step DenyApproval required leave {
          execute = DenyOrderApproval;
        }
      }
    if (object.preapproved) {
      activity {
        actor {
          pos = DeptManager;
          orgunit = OrderProcessing;
        }
        step ApproveOrder required leave {
          execute = ApproveOrder;
        }
        step DenyApproval required leave {
          execute = DenyOrderApproval;
        }
      }
    }
    // This process may only be used as a sub process, and may not be
    // instantiated directly on an object
    procdefstate = PROCDEF_SUBPROCESS;
  }
  process ApproveOrderWF {
    parallel {
      activity {
        actor {
          pos = Clerk;
          orgunit = OrderProcessing;
        }
        step ApproveOrder required leave {
          execute = ApproveOrder;
        }
        step DenyApproval required leave {
          execute = DenyOrderApproval;
        }
      }
      // Embedding sub process APPDUCXSAMPLE@200.200:DeptMgrApprovalWF
      activity {
        subproc = DeptMgrApprovalWF;
```

```
        }
      }
    }
}
```

10.6 BPMN2 Modeling with app.ducx

Fabasoft app.ducx allows you to import a BPMN2 process definition into your Fabasoft app.ducx Project via an import wizard using “File” – “Import”. In the category “Fabasoft app.ducx” you will find an item called “BPMN2 Process Definition into Fabasoft app.ducx Project”.

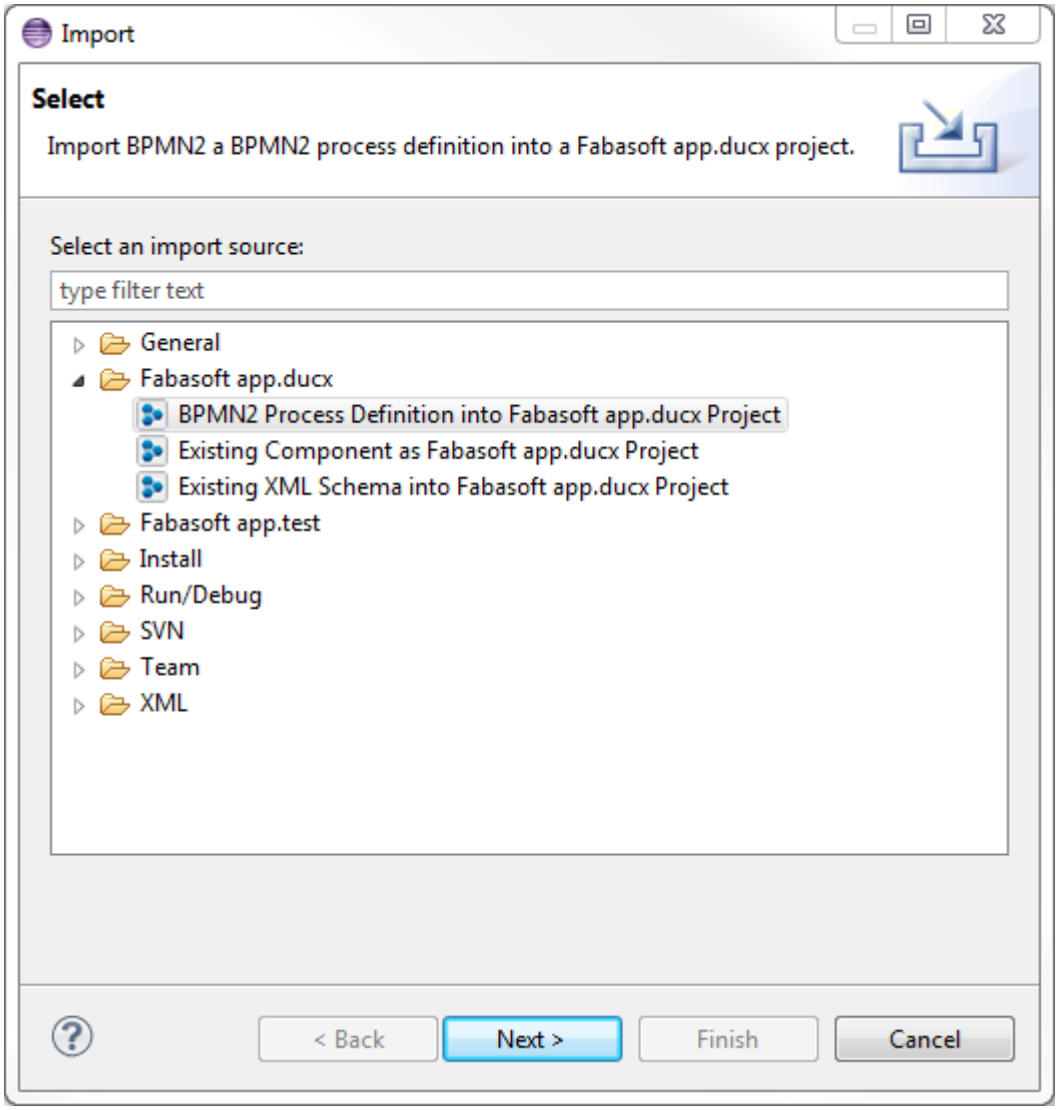


Figure 22: Wizard for importing a BPMN2 Model (1)

On the next page, you can either select a process definition form your local file system or a process definition from Folio Cloud or your “Default Webservice”.

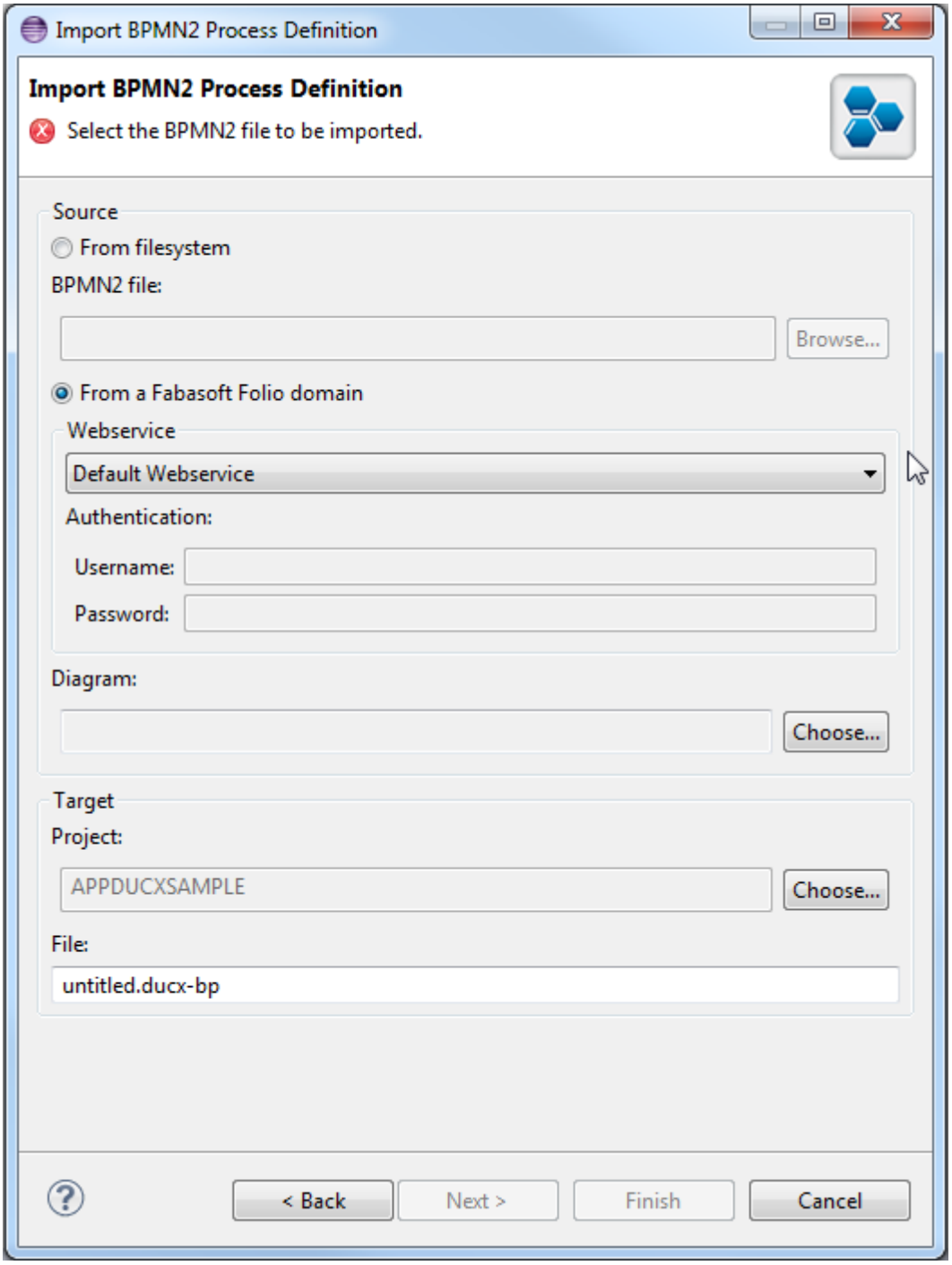


Figure 23: Wizard for importing a BPMN2 Model (2)

By finishing this dialog, the process definition will be transformed to a Fabasoft app.ducx compatible business process which you can afterwards customize or use.

Note:

- If your BPMN2 process model uses references to predefined activity definitions it is necessary that the software components of these references are already added to the “Software Component References” of your Fabasoft app.ducx Project.

11 app.ducx Customization Language

The purpose of the app.ducx customization language is to define, customize and tailor your software component to project- or solution-specific requirements.

A customization block consists of import declarations, customization points and customizations. The `customization` keyword denotes a customization model block. It must be followed by the reference of your software component and curly braces.

Syntax

```
customization softwarecomponent
{
  // Import declarations
  import softwarecomponent;
  ...
  // Customization point
  CustomizationPointReference(parameter, ...);
  ...
  // Customization for the customization point
  customize CustomizationPointReference<keyparameter, ...> {
    ...
  }
  ...
}
```

11.1 Customization points

Customization points are defined within a customization model block. To declare a customization point, define the reference of the customization point followed by parentheses holding the list of parameters. The declaration is finished with a semicolon.

Example

```
GetObjHint
(
  key ObjectClass objclass,
  string suffix,
  retval string hint
);


GetAllowedAttrDef
(
  key ObjectClass objclass,
  key AttributeObjectDef attrdef,
  out direct AttributeObjectDef outattrdef
);
```

- Parameters denoted with the keyword `key` are used to define which customization applies to which objects. Keys may be marked as optional (`optional key`). So it is possible to let the optional key empty, if edited via the user interface.
- All input parameters are available in expression blocks (local scope) of customizations.
- Output parameters (`out`, `retval`) can be defined as `direct`. Values can be assigned directly to direct parameters (`hint = "string directly assigned"`), otherwise values can only be assigned using an expression block (`hint = expression { "expression assignment" }`).

11.1.1 Defining domain types

With help of *Domain Types* (`COOSYSTEM@1.1:DomainType`) it is possible to define configurations and ACLs abstractly for domain types instead of concrete domains. This can be achieved by using the keyword `instance` followed by the object class `DomainType` and the name of the instance reference.

Syntax

```
objmodel APPDUCXSAMPLE@200.200
{
  // Import declarations
  import softwarecomponent;
  ...
  // Creates an instance of a domain type
  instance DomainType DomainTypeReference {
    configurationreference = configurationobjectreference
    ...
  }
  ...
}
```

All configurations that are defined in the domain type instance can be used for customization points in foreign software components.

To reuse a defined configuration in your own software component, the configuration object instance has to be created manually. The reference of the configuration object instance has to comply with the following rule: reference of the domain type followed by the software component of the configuration followed by “Config” (e.g. `DomainTypeAppducxSampleFSCCONFIGConfig`).

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCSMTP@1.1001;
  import FSCCONFIG@1.1001;

  instance AdministrationConfiguration DomainTypeAppducxSampleFSCCONFIGConfig {}

  instance DomainType DomainTypeAppducxSample {
    clientsmtpconfig = DefaultConfiguration;
    clientconfiguration = DomainTypeAppducxSampleFSCCONFIGConfig;
  }
}
```

11.1.2 Generic method implementation for customization points

If a customization point needs additionally an implementation of a specified action with a specified method implementation for the object class defined in a customize statement, this can be defined in the customization point to reduce the effort in the customizing statements.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  extend instance NameBuild {
    typecpmethods<cpaction,cpmethdefinition> = {
      {
        EvaluateGenericNameBuild,
        MethodGenericNameBuild
      }
    }
  }
```

```
}
```

If this customization point is used for an object class in a customizing statement, the mapping is now created implicitly.

11.2 Customizations

For each customization point several customizations may exist, distinguished by the `key` parameters. Optional keys may be set to `null`, if the customization should apply to any value in the optional key. The customization provides the concrete implementation of a customization point.

By default customizations are implicitly stored in the default configurations of the software components the corresponding customization point belongs to. If you create your own customization points a configuration with reference `DefaultConfig` is implicitly generated for your software component.

11.2.1 Using domain types

If a Fabasoft Folio Domain consists of several Fabasoft Folio tenants you might want to have an own configuration for each Fabasoft Folio Tenant. This can be achieved by defining a software solution (or software edition) for each Fabasoft Folio Tenant and using the `target` keyword followed by the reference of the domain type.

A domain type can be customized if:

- A configuration object comply with the following rule: reference of the domain type followed by the software component of the configuration followed by “Config” (e.g. `DomainTypeAppducxSampleFSCCONFIGConfig`)
- Only the domain type without configuration exists but it is defined in the own software component. Then a configuration will be created and the domain type extended.
- The customization point is defined in the own software component, but the configuration object is not yet defined. Then a configuration will be created and the domain type extended.
- The customization point is defined in a component, which added the own software component as a friend, but the configuration object is not yet defined. Then a configuration will be created and the domain type extended.

In all other cases an error will be thrown.

Example

```
// Define the customization for the own software solution
// Create a new configuration object if it is not already created manually
target DomainTypeAppducxSample {
  customize GetAllowedAttrDef<Folder, objchildren> {
    // Assumes that outattrdef is defined as direct
    outattrdef = objsubject;
  }
}

// Define the customization for the software edition FolioCloud
// Add an entry to an existing configuration object in the FolioCloud configuration
target DomainTypeEditionFolioCloud {
  customize GetAllowedAttrDef<Folder, objchildren> {
    outattrdef = objsubject;
  }
}
```

11.2.2 Deprecated: Using add and override of a software solution or software edition

If a Fabasoft Folio Domain consists of several Fabasoft Folio tenants you might want to have an own configuration for each Fabasoft Folio Tenant. This can be achieved by defining a software solution (or software edition) for each Fabasoft Folio Tenant and using the `target` keyword followed by the reference of the software solution (or software edition) and the keyword `add` or `override`. If the keyword `add` is used, the customization is added to an existing configuration. If the keyword `override` is used, a new configuration gets generated. The configurations have to be assigned to the Fabasoft Folio Tenant manually via the Fabasoft Folio Web Client (“Domain Administration” > “Object List” > “Domain Objects”).

Cloud profile note: For `EditionFolioCloud@1.1`, only addition is allowed, override is forbidden for all software solutions and editions.

Example

```
// Implementation for Object
customize GetObjHint<Object> {
  // Assumes that hint is not defined as direct
  // The parameter suffix is available in the expression block
  hint = expression { cooobj.objname + " " + suffix }
}
// Implementation for ContentObject
customize GetObjHint<ContentObject> {
  // Assumes that hint is not defined as direct
  hint = expression { cooobj.objname }
}
customize GetAllowedAttrDef<Folder, objchildren> {
  // Assumes that outattrdef is defined as direct
  outattrdef = objname;
}
// Define the customization for a specific software solution
// Create a new configuration object (override)
target SolutionPayment@1.1 override {
  customize GetAllowedAttrDef<Folder, objchildren> {
    // Assumes that outattrdef is defined as direct
    outattrdef = objsubject;
  }
}
```

11.3 Using customizations

Customizations can be used in expression blocks.

Example

```
// The result is assigned to @tmp
impl = expression {
  cooobj.GetObjHint(cooobj.objclass, @suffix, &@tmp);
}
// build is declared as retval, thus the result can be assigned directly
impl = expression {
  string @tmp = cooobj.GetObjHint(cooobj.objclass, @suffix);
}
// The result is a list of matching customizations assigned to @result
impl = expression {
  GetAllowedAttrDef[] @result = cooobj. GetAllowedAttrDef(cooobj.objclass, null)[...];
}
```

Note:

- The key parameters are used to evaluate which customization should be used. If a parameter is an object class, the inheritance is taken into account (best match).
- If a parameter is defined as `retval` direct assignments are possible.
- If an output parameter is omitted, its value is not calculated due to performance reasons.
- If you need to access all matching configuration entries you can access this list by using "[…]" to qualify the result of the customization point.

11.3.1 Concise example

This concise example subsumes the concepts described in the above chapters. The goal is to customize the hint that is displayed when moving the mouse over a content object.

Example

app.ducx Customization Language

```
GetObjHint
(
  key ObjectClass objclass,
  string suffix,
  retval string hint
);
customize GetObjHint<ContentObject> {
  hint = expression {
    return
      cooobj.GetHTMLLine(#objname, cooobj.objname) +
      cooobj.GetHTMLLine(#objowner, cooobj.objowner.objname);
  }
}
```

app.ducx Use Case Language

```
// Override COOATTREDIT@1.1:GetObjectHint
override GetObjectHint {
  variant ContentObject {
    impl = expression {
      // Call customization point
      cooobj.GetObjHint(cooobj.objclass, null, &text);
    }
  }
}
GetHTMLLine(AttributeDefinition attrdef, any value, retval string line) {
  variant Object {
    impl = expression {
      line  = "<b>";
      line += attrdef.objname;
      line += ":</b> ";
      line += STRING(value);
      line += "</br>";
      return line;
    }
  }
}
```

11.4 Predefined customization points

Several customization points are defined by the base product itself. In the following chapters these customization points are explained. The following examples are mainly based on Fabasoft Folio.

11.4.1 PreGUI

The customization point `PreGUI` allows initializing values before the constructor form is displayed.

Syntax

```
customize PreGUI<objectclass> {
  steps = expression {...}
}
```

Description:

- *objectclass*
  The customization applies to objects of the defined object class.
- steps
  Defines a Fabasoft app.ducx Expression that is executed.

Example

app.ducx Object Model Language

```
instance ComponentState StateCaptured {
}
instance ComponentState StateApproved {
}
instance ComponentState StateRejected {
}
instance ComponentDocumentCategory PreGUICategory {
  dcshortform = "PreGui";
  dcstate = { StateCaptured,
              StateApproved,
              StateRejected }
}
```

app.ducx Customization Language

```
customize PreGUI<Incoming> {
  steps = expression {
    this.COOSYSTEM@1.1:ObjectLock(true, true);
    this.FSCFOLIO@1.1001:bostate = #StateCaptured;
    this.FSCFOLIO@1.1001:InitPreGUI();
    this.FSCFOLIO@1.1001:InitWithCategory(#PreGUICategory);
  }
}
```

Note: In the Fabasoft Folio Domain the customization gets defined in the template configuration.

11.4.2 InitWithState

The customization point `InitWithState` is a specialization of the customization point `PreGUI`.

Syntax

```
customize InitWithState<objectclass, state, category> {}
```

Description:

- *objectclass*
  The customization applies to objects of the defined object class.
- *state*
  Defines a state that is assigned to `FSCFOLIO@1.1001:bostate`.

- *category*
  Defines a category that is used for `FSCFOLIO@1.1001:InitWithCategory(#category)`.

Example

```
customize InitWithState<Incoming, StateCaptured, PreGUICategory> {}
```

Note: In the Fabasoft Folio Domain the customization gets defined in the template configuration.

11.4.3 PostGUI

The customization point `PostGUI` allows executing a Fabasoft app.ducx Expression after the user clicked “Next” or “Apply” of a constructor form.

Syntax

```
customize PostGUI<objectclass> {
  steps = expression {...}
}
```

Description:

- *objectclass*
  The customization applies to objects of the defined object class.
- steps
  Defines a Fabasoft app.ducx Expression that is executed.

Example

```
customize PostGUI<NoteObject> {
  steps = expression {
    this.COOSYSTEM@1.1:ObjectLock(true, true);
    this.COOSYSTEM@1.1:objsubject = "MySubject";
  }
}
```

Note: In the Fabasoft Folio Domain the customization gets defined in the template configuration.

11.4.4 IncreaseOrdinal

The customization point `IncreaseOrdinal` allows increasing a key numerator at `PostGUI`.

Syntax

```
customize IncreaseOrdinal<objectclass, property> {}
```

Description:

- *objectclass*
  The customization applies to objects of the defined object class.
- *property*
  Defines the numerator property that should be used.

Example

```
class ObjOrdinal : Document {
  KeyNumerator ordinal {
    /*
     * Specifies amount of preallocation
     */
    allocamount = 101;
    KeyEntryList<NUMERATOR@1.1001:objclass, keyattrlist> = {
      { ObjOrdinal, { documentyear } }
    }
  }
}
```

app.ducx Customization Language

```
customize IncreaseOrdinal<ObjOrdinal, ordinal> {}
```

Note: The parameter `alloc` has a default value of `true`. To create a numerator without preallocation you must explicitly specifiy `alloc = false`.

Note: In the Fabasoft Folio Domain the customization gets defined in the template configuration. This means that overrides objects from `FSCVTC@1.1001` for all applications responsible for creation of must be provided:

Example

```
// Application executed when an object is created via menu or
// button in an object list
override FSCVENV@1.1001:InitializeCreatedObject {
  variant objectclass {
    impl = FSCVTC@1.1001:InitializeCreatedObjectApp;
  }
}
// Application executed when an object is created inside of
// an object pointer property
override FSCVENV@1.1001:InitializeCreatedObjectDoDefault {
  variant objectclass {
    impl = FSCVTC@1.1001:InitializeCreatedObjectApp;
  }
}
// Application executed when an object is created from
// a template
override FSCVENV@1.1001:InitializeTemplateCreatedObject {
  variant objectclass {
    impl = FSCVTC@1.1001:InitializeCreatedObjectApp;
  }
}
```

11.4.5 FormatValue

The customization point `FormatValue` allows formatting values. For the defined object class, the action `COOSYSTEM@1.1:ObjectPrepareCommit` will be overwritten implicitly.

Syntax

```
customize FormatValue<objectclass, trigger> {
  build = expression {...}
}
```

Description:

- *objectclass*
  Defines the object class of the object that contains the value that should be formatted.

- *trigger*
  Defines the property that should be formatted.
- build
  Defines a Fabasoft app.ducx Expression that formats the value.

Example

```
customize FormatValue<Project, objname> {
  build = expression {
    this.boshortform + " " + Format(this.recnumber, "0000")[4] + " - "
  }
}
```

Note: In the Fabasoft Folio Domain the customization gets defined in the administration configuration.

11.4.6 MetaParticipant

The customization point `MetaParticipant` defines how the meta participant is resolved.

Syntax

```
customize MetaParticipant<objclass, metapart, object, activity, process, participant> {
  newparticipant = expression {...}
}
```

Description:

- *objclass*
  The customization applies to objects of the defined object class.
- *metapart*
  Defines the meta participant that should be resolved.
- *object*
  The object, on which the process is running.
- *activity*
  The current activity instance.
- *process*
  The object of the current process.
- *participant*
  The current participant of the activity.
- newparticipant
  Defines the expression that is used to resolve the meta participant to a participant.

Example

```
customize MetaParticipant<Object, WFMP_INITIATOR> {
  newparticipant = expression {
    @owner = object.objowner;
    COOWF@1.1:WorkFlowParticipant @part = {};
    if (@owner != null) {
      @roles = @owner.userroles[COOSYSTEM@1.1:default==true];
      if (count(@roles) > 0) {
        @role = @roles[0];
        @rolegroup = @role.userrolegroup;
        if (@rolegroup != null) {
          if (@role.userrolepos == #FSCFOLIO@1.1001:HeadPos) {
```

```
            @supergroups = @rolegroup.grsupergroups;
            if (count(@supergroups) > 0) {
              @part.COOWF@1.1:wfpgroup = @supergroups[0];
            }
            else {
              @part.COOWF@1.1:wfpgroup = @rolegroup;
            }
            @part.COOWF@1.1:wfpposition = #FSCFOLIO@1.1001:HeadPos;
          }
          else {
            @part.COOWF@1.1:wfpgroup = @rolegroup;
            @part.COOWF@1.1:wfpposition = #FSCFOLIO@1.1001:HeadPos;
          }
        }
        else {
          @part.COOWF@1.1:wfpmetaparticipant =
            participant.COOWF@1.1:wfpmetaparticipant;
        }
      }
      else {
        @part.COOWF@1.1:wfpmetaparticipant =
          participant.COOWF@1.1:wfpmetaparticipant;
      }
    }
    else {
      @part.COOWF@1.1:wfpmetaparticipant =
        participant.COOWF@1.1:wfpmetaparticipant;
    }
    @part;
  }
}
```

Note: In the Fabasoft Folio Domain the customization gets defined in the workflow configuration.

11.4.7 ACLConfiguration

The customization point `ACLConfiguration` allows defining tenant specific ACLs that should be assigned to objects. The ACL is evaluated using following steps. The first step that returns an ACL defines the used ACL. Step 1 and 4 can be configured by the customization point.

1. Use a tenant specific ACL for an object, if a customization is defined for the corresponding object class (object class hierarchy gets considered).
   Trigger: `COOSYSTEM@1.1001:classdefaultacl`
2. Use an ACL that is defined for the object class (ACLs created in a corresponding tenant are preferred).
3. Use an ACL that is defined in the group of the user (*Default ACL for New Objects*, *ACL Objects*).
4. Use a tenant specific default ACL for an object, if a customization is defined for the corresponding object class (object class hierarchy gets considered).
   Trigger: `COOSYSTEM@1.1001:objaclobj`
5. Hard-coded: If the steps before do not return an ACL, finally one of these ACLs is used based on the object class: “Default ACL”, “ACL for Administration Objects” or “ACL for Developer Objects”.

Syntax

```
customize ACLConfiguration<cfgobjclass, trigger> {
  acl = ...
}
```

Description:

- *cfgobjclass*
  The customization applies to objects of the defined object class.
- *trigger*
  If step 1 should be customized, `COOSYSTEM@1.1001:classdefaultacl` has to be used. If step 4 should be customized, `COOSYSTEM@1.1001:objaclobj` has to be used.
- `acl`
  Defines the ACL that should be assigned.

Example

```
customize ACLConfiguration<Folder, classdefaultacl> {
  acl = COOSYSTEM@1.1:DefaultGlobalACL;
}
```

Note: In the Fabasoft Folio Domain the customization gets defined in the administration configuration.

11.4.8 ImageTypeConfiguration

The customization point `ImageTypeConfiguration` allows defining parameters for thumbnails and preview images.

Syntax

```
customize ImageTypeConfiguration<imgclass, imgtype> {
  imgwidth = ...
  imgheight = ...
  imgformat = ...
  imgpages = ...
  imgdstattrdef = ...
  imgdefaultthumbexpr = expression {...}
}
```

Description:

- *imgclass*
  The customization applies to objects of the defined object class.
- *imgtype*
  Defines whether the customization applies to thumbnails (“tn”) or preview images (“pv”).
- `imgwidth`
  Defines the maximum width in pixel of the displayed image (e.g. 512).
- `imgheight`
  Defines the maximum height in pixel of the displayed image (e.g. 256).
- `imgformat`
  Defines the format of the displayed image (default: “jpg”; possible values: “gif”, “png”).
- `imgpages`
  Defines how many pages of a document should be available as image.
- `imgdstattrdef`
  Defines the property where the image should be cached (e.g. `COODESK@1.1:objthumbnailimage` or `COODESK@1.1:objpreviews`).
- `imgdefaultthumbexpr`
  Defines an expression that is used to evaluate a fallback image, if the actual image is not available (e.g. `cooobj.FSCVENV@1.1001:GetObjectImageFallback(imgtype)`).

Example

```
customize ImageTypeConfiguration<Object, "tn"> {
  imgwidth = 512;
  imgheight = 256;
  imgdstattrdef = objthumbnailimage;
  imgdefaultthumbexpr = expression {
    cooobj.GetObjectImageFallback(imgtype)
  }
}
```

Note: In the Fabasoft Folio Domain the customization gets defined in the conversion configuration.

11.4.9 ContentConfiguration

The customization point `ContentConfiguration` defines in which property the desired content can be found (e.g. content for the thumbnail or preview image generation).

Syntax

```
customize ContentConfiguration<contclass> {
  contsrcexpr = expression {...}
}
```

Description:

- *contclass*
  The customization applies to objects of the defined object class.
- contsrcexpr
  Defines the expression that is used to evaluate the desired content (e.g. for the thumbnail and preview image generation or for conversion tasks).

Example

```
customize ContentConfiguration<ContentObject> {
  contsrcexpr = expression { cooobj.classinitcont.contcontent }
}
```

Note: In the Fabasoft Folio Domain the customization gets defined in the conversion configuration.

11.4.10 CPQuickSearchAction

The customization point `CPQuickSearchAction` allows overriding the action to be called to perform the quick search in an object pointer property.

Syntax

```
customize CPQuickSearchAction<cfgobjclass, cfgview> {
  cfgqsaction = ...;
}
```

Description:

- *cfgobjclass*
  The customization applies to objects of the defined object class.
- *cfgview*
  The property, to which the customization applies when a quick search is executed.

- `cfgqsaction`
  Action to be called as quick search action.

The following example defines `SearchUser` as action for the quick search in the owner property.

Example

```
customize CPQuickSearchAction<Object, objowner> {
  cpqsaction = SearchUser;
}
```

11.4.11 CPQuickSearchAppearance

The customization point `CPQuickSearchAppearance` defines for which object class in which object pointer property the enhanced or simple appearance is used.

Syntax

```
customize CPQuickSearchAppearance<cfgobjclass, cfgattrdefopt> {
  appearance = ...;
}
```

Description:

- *`cfgobjclass`*
  The customization applies to objects of the defined object class.
- *`cfgattrdefopt`*
  The property, to which the customization applies when a quick search is executed.
- `appearance`
  Defines whether the results of the quick search in the property `property` are displayed “Enhanced” or “Simple”.

The following example defines the enhanced appearance for all objects, which are searched in the *Owner* (`COOSYSTEM@1.1:objowner`) property.

Example

```
customize CPQuickSearchAppearance<Object, objowner> {
  appearance = QS_ENHANCED;
}
```

11.4.12 CPQuickSearchSuffix

The customization point `CPQuickSearchSuffix` defines the properties to be displayed in the enhanced appearance. A search result of the object class `cfgobjclass` is displayed in the `cfgattrdefopt` list according to the `additionaldescription` and the `cfgmultilinedescription` expression. If `cfgattrdefopt` does not contain a value, the setting applies to all lists.

For the “Simple” appearance the result of the `cfgmlnamesuffix` expression is used. For the “Enhanced” appearance the result of the `cfgmultilinedescription` expression is used. If this expression is not defined, the `cfgmlnamesuffix` expression is used.

Syntax

```
customize CPQuickSearchSuffix<cfgobjclass, cfgattrdefopt > {
  cfgmlnamesuffix = expression {...}
  cfgmultilinedescription = expression {...}
}
```

Description:

- *cfgobjclass*
  The customization applies to objects of the defined object class.
- *cfgattrdefopt*
  Optionally the customization applies to this property.
- cfgmlnamesuffix
  This result can only be displayed in one line and is displayed after the object name.
- cfgmultilinedescription
  This Fabasoft app.ducx expression defines the additional properties to be displayed. The result can also be displayed in multiple lines. For displaying a multiline result \n is used for starting a new line.

The following example defines that for search results of the *User* (COOSYSTEM@1.1:User) object class also the default role is displayed.

Example

```
customize CPQuickSearchSuffix<User> {
   cfgmultilinedescription = expression {
     UserRoleList[] @defroles = cooobj.userroles[COOSYSTEM@1.1:default];
     string @rolename = "";
     if (count(@defroles) == 0) {
       @defroles = cooobj.userroles;
     }
     if (count(@defroles) > 0) {
       UserRoleList @defrole = @defroles[0];
       if (@defrole != null) {
         @rolename = @defrole.userrolepos.objname;
         @rolename += " "+@defrole.userrolegroup.grshortname;
       }
     }
     [cooobj.objname, @rolename];
   }
}
```

11.4.13 CPSymbols

The customization point CPSymbols allows defining the symbol of objects from a certain object class.

The customization point gets evaluated in the FSCCONFIG@1.1001:MethodGenericIconGet method. This method can be used for the COODESK@1.1001:AttrObjMiniIconGet action, which has to be implemented for the appropriate object class (cfgobjclass).

Syntax

```
customize CPSymbols<cfgobjclass> {
  cfgicexpression = expression {...}
  objmicon = ...;
}
```

Description:

- *cfgobjclass*
  The customization applies to objects of the defined object class.
- cfgicexpression
  Defines a Fabasoft app.ducx Expression that has to apply.

- `objmicon`
  Defines the symbol that is assigned to `COODESK@1.1:Symbol`.

Example

```
customize CPSymbols<User> {
  cfgicexpression = expression {
    return true;
  }
  objmicon = SymbolUserAccepted;
}
```

Note: In the Fabasoft Folio Domain the customization gets defined in the administration configuration.

11.4.14 CPValidationExpression

The customization point `CPValidationExpression` is used for the validation of an entered value. The customization point gets evaluated in the `FSCCONFIG@1.1001:MethodValidateSet` method, which can be called in the `FSCCONFIG@1.1001:AttrValidateSet` action.

The local scope contains the current object and the global scope contains a dictionary with the old value and the new value.

Syntax

```
customize CPValidationExpression<cfgobjclass, trigger, context> {
  cfgexpression = expression {...}
  errorobject = ...;
}
```

Description:

- *`cfgobjclass`*
  The customization applies to objects of the defined object class.
- *`trigger`*
  The key for the evaluation of the output parameter.
- *`context`*
  The optional key for the evaluation of the output parameter.
- `cfgexpression`
  If this expression returns “true”, the error (`errorobject`) is displayed. Alternatively the expression itself can display an error.
- `errorobject`
  The error message that is displayed in case of an error.

Example

```
customize CPValidationExpression<User, DialogUserPasswordForgot, objname> {
  cfgexpression = expression {
    return true;
  }
  errorobject = ErrActivateUserFound;
}
```

Note: In the Fabasoft Folio Domain the customization gets defined in the administration configuration.

11.4.15 CPContextExpressions

The customization point `CPContextExpressions` returns an expression, depending on the input parameters `trigger` and `context`.

The customization point gets evaluated in the `FSCCONFIG@1.1001:EvaluateExpression` action. This action has to be called for the appropriate object class (`cfgobjclass`) with the input parameters `trigger` and optionally `context`.

Syntax

```
customize CPContextExpressions<cfgobjclass, trigger, context> {
  cfgexpression = expression {...}
}
```

Description:

- *cfgobjclass*
  The customization applies to objects of the defined object class.
- *trigger*
  The key for the evaluation of the output parameter.
- *context*
  The optional key for the evaluation of the output parameter.
- cfgexpression
  The expression to be returned.

Example

```
customize CPContextExpressions<User, DialogUserPasswordForgot, objname> {
  cfgexpression = expression {
    return true;
  }
}
```

Note: In the Fabasoft Folio Domain the customization gets defined in the administration configuration.

11.4.16 CPDocStateValidateConfig

The customization point `CPDocStateValidateConfig` is used for validating state changes of documents.

To evaluate the customization point, the `FSCFOLIO@1.1001:ValidateDocStateChange` action has to be called for an object of the appropriate object class (`dsvcobjclass`). By means of the state change (`dsvcoldstate`, `dsvcnewstate`) this action determines the respective expression (`dsvcvalidateexpr`) and evaluates it. Optionally the defined signature type (`dsvcsigntype`) can be executed.

Syntax

```
customize CPDocStateValidateConfig<dsvobjclass, dsvcoldstate, dsvcnewstate> {
  dsvcvalidateexpr = expression {...}
  dsvcsigntype = ...;
}
```

Description:

- *dsvcobjclass*
  The customization applies to objects of the defined object class.
- *dsvcoldstate*
  The state of the document, before the state change.
- *dsvcnewstate*
  The state of the document, after the state change.
- dsvcvalidateexpr
  An optional Fabasoft app.ducx Expression that has to be apply.
- dsvcsigntype
  Defines the *Signature Type* (FSCFOLIO@1.1001:dsvcsigntype) that has to be used after the state changed.

Example

```
customize CPDocStateValidateConfig<Case, DS_EDIT, DS_CLOSED> {
  dsvcvalidateexpr = expression {
    return true;
  }
  dsvcsigntype = SIGN_CLOSE;
}
```

Note: In the Fabasoft Folio Domain this customization gets defined in the Folio configuration.

11.4.17 CPRestrictClasses

The customization point CPRestrictClasses is used to restrict createable and searchable classes. In an expression all restricted classes are returned.

Syntax

```
customize CPRestrictClasses<cfgobjclass, cfgattrdef, cfgcamode> {
  cfgcpexpression = expression {...}
}
```

Description:

- *cfgobjclass*
  The customization applies to objects of the defined object class.
- *cfgattrdef*
  The property, to which the customization applies.
- *cfgcamode*
  The validation mode for allowed classes.
- *cfgcpexpression*
  The expression to be returned. This expression contains restricted classes.

Example

```
customize CPRestrictClasses <Case, DS_EDIT, DS_CLOSED> {
  cfgcpexpression = expression {
    return true;
  }
}
```

Note: For the expression, the values attrdef, mode, restriction, allowedclasses, notallowedclasses are available in the global scope.

11.4.18 NameBuild

The customization point `NameBuild` is used to format values.

Syntax

```
customize NameBuild<cfgobjclass> {
  properties = {...}
  build = expression {...}
  format = stringvalue
  namefixed = booleanvalue
}
```

Description:

- *cfgobjclass*
  The customization applies to objects of the defined object class.
- properties
  The properties, to which the customization applies.
- *build*
  Defines a Fabasoft app.ducx Expression that formats the value.
- *format*
  Defines a string object with a multilingual name.
- *namefixed*
  Defines if the value is editable.

Example

```
customize NameBuild <Account> {
  properties = {
    accountnumber,
    usersurname
  }
  build = expression {
    return cooobj.accountnumber + " / " + cooobj.usersurname;
  }
  namefixed = true;
}
```

Note: To use this customization point with Fabasoft domains until 2012 Summer Release, it is necessary to implement `FSCCONFIG@1.1001:EvaluateGenericNameBuild` for the class defined in `cfgobjclass` with `FSCCONFIG@1.1001:MethodGenericNameBuild`.

11.4.19 FilterDispViewListAction

The customization point `FilterDispViewListAction` determines the filter action for the display view settings for the given object class.

Syntax

```
customize FilterDispViewListAction<objclass> {
  filteraction = expression {...}
}
```

Description:

- *objclass*
  The customization applies to objects of the defined object class.

- *filteraction*
  This action is used to verify or override the resulting display view list. The action uses the prototype `COODESK@1.1:FilterDispViewListPrototype`.

Example

app.ducx Customization Language

```
customize FilterDispViewListAction<Folder> {
  filteraction = expression {
    return #MYCUSTOMIZE@1.1065:FolderDispViewListFolder
  }
}
```

app.ducx Use Case Language

```
usecase FolderDispViewListFolder(parameters as FilterDispViewListPrototype) {
  variant Folder {
    impl = expression {
      if (writelocation >= 0) {
        // write columns: prevent saving settings
        throw #COOSTERR_CANCEL;
      }
      else {
        // read columns: fixed set of display columns
        DisplayColumnList[] mycolumns;
        mycolumns += coort.CreateAggregate(#DisplayColumnList);
        mycolumns += coort.CreateAggregate(#DisplayColumnList);
        mycolumns += coort.CreateAggregate(#DisplayColumnList);
        mycolumns[0].dispattribute = #objname;
        mycolumns[1].dispattribute = #objcreatedby;
        mycolumns[2].dispattribute = #objaclobj;
        displaylist.dispcolumns = mycolumns;
      }
    }
  }
}
```

11.4.20 AggregationOverride

The customization point `AggregationOverride` replaces an aggregation action for the given object class and is available in the software component `FSCVENVUI@1.1001`.

Syntax

```
customize AggregationOverride<objclass, attrdef, aggract> {
  aggroverride = expression {...}
}
```

Description:

- *objclass*
  The customization applies to objects of the defined object class.
- *attrdef*
  The property, to which the customization applies.
- *aggract*
  The aggregation action, which is to be overridden.
- *aggroverride*
  This action is used to override the aggregation action. The action uses the prototype `COODESK@1.1:AggregationPrototype`.

Example

app.ducx Customization Language

```
customize AggregationOverride<Object, objcontsize, GetSum> {
  aggroverride = expression {
    //calculates the sum of the values and adds a "KB" suffix
    return #FSCVENVUI@1.1001:GetSumContSize
  }
}
```

11.4.21 InsertActivityDef

The customization point `InsertActivityDef` calculates an activity definition for a specified object class with a supplied context.

Syntax

```
customize InsertActivityDef<objclass, scope> {
  insertactdef = expression {...}
}
```

Description:

- *objclass*
  The customization applies to objects of the defined object class.
- *scope*
  A component object, that is used in combination with the object class to retrieve an activity definition.
- *insertactdef*
  Defines a Fabasoft app.ducx Expression to calculate the activity definition. Following values are available in the local scope:
  - activity: The currently used activity instance of the current process
  - process: The currently used process instance
  - participant: The participant of the current activity instance
  - object: The object of the current process

Example

app.ducx Customization Language

```
customize InsertActivityDef<Object, SIGN_APPROVAL_ACCEPTED> {
  insertactdef = expression {
    return #ActDefApprovalAccepted;
  }
}
```

Note: In the Fabasoft Folio Domain this customization gets defined in the workflow configuration.

11.4.22 GetLogoContainer

The customization point `GetLogoContainer` defines which logo is displayed based on the object class of the selected object.

Syntax

```
customize GetLogoContainer<objclass> {
  cont = expression {...}
}
```

Description:

- *objclass*
  The customization applies to objects of the defined object class.
- *cont*
  Defines a Fabasoft app.ducx Expression to calculate the object that contains the logo (`COODESK@1.1:objlogoimage`) that should be displayed.

Example

app.ducx Customization Language

```
customize GetLogoContainer<Object> {
  cont = expression {
    if (cooobj.HasClass(#TeamRoom)) {
      cooobj;
    }
    else {
      cooobj.objteamroom != null ? cooobj.objteamroom : null;
    }
  }
}
```

11.4.23 CreatePortalConfiguration

The customization point `CreatePortalConfiguration` defines the template for creating a portal entry if an object is dragged to the portal list.

Syntax

```
customize CreatePortalConfiguration<cfgobjclass> {
  portalentry = ….;
}
```

Description:

- *cfgobjclass*
  The customization applies to objects of the defined object class.
- *portalentry*
  Defines the template to be used to create the portal entry.

Example

app.ducx Customization Language

```
customize CreatePortalConfiguration<Object> {
  portalentry = PaneDesk;
}
```

11.4.24 CPAllowedAttrDef

The customization point `CPAllowedAttrDef` allows redefining object pointer properties in the current domain for checking and filtering allowed object classes.

Syntax

```
customize CPAllowedAttrDef<cfgobjclass,cfgattrdef,cfgcamode> {
  cfgrefattrdef = ….;
  cfgexpression = ….;
}
```

Description:

- *cfgobjclass*
  The customization applies to objects of the defined object class.
- *cfgattrdef*
  References the object pointer property definition to be redefined.
- *cfgcamode*
  Defines the mode for checking or filtering allowed objectclasses.
- *cfgrefattrdef*
  Defines the new object pointer property for checking or filtering allowed object classes.
- *cfgexpression*
  Conditional expression to be executed to check if the replace has to be done.

Example

app.ducx Customization Language

```
customize CPAllowedAttrDef<Story,objchildren,null> {
  cfgrefattrdef = scrumdocuments;
}
```

11.4.25 ListOption

The customization point `ListOption` allows overriding the SimList control parameters.

Syntax

```
customize ListOption<objclass,attrdef> {
  createapplication = expression {...}
  createsymbol = expression {...}
  createtext = expression {...}
  searchtext = expression {...}
  typetext = expression {...}
  showicon = expression {...}
  showhover = expression {...}
  showunderline = expression {...}
  wildcardsearch = expression {...}
}
```

Description:

- *objclass*
  The customization applies to objects of the defined object class.
- *attrdef*
  References the object pointer property definition to be redefined.
- *createapplication*
  Defines an expression to get the application for creating objects in the control.
- *createsymbol*
  Defines an expression to get the symbol for creating objects in the control.
- *createtext*
  Defines an expression to get the description for creating objects in the control.
- *searchtext*
  Defines an expression to get the search text for creating objects in the control.

- *typetext*
  Defines an expression to get the data entry instruction of the control.
- *showicon*
  Defines an expression to set a symbol for an entry.
- *showhover*
  Defines an expression to set a hover text for an entry.
- *showunderline*
  Defines an expression to set an unterline representation for an entry.
- *wildcardsearch*
  Defines an expression to allow wild card search in the control.

Example

app.ducx Customization Language

```
customize ListOption<TransmissionLog,tlinvitedusers> {
  createapplication = expression {
    return #CreateObjectApp;
  }
  createsymbol = expression {
    return (cootx.isCreated(cooobj) ? #SymbolAdd : #MiniIconObjectCreate );
  }
  createtext = expression {
    return (cootx.isCreated(cooobj) ? #StrAddFavorite : #StrCreate);
  }
  typetext = expression {
    return #StrTypeText;
  }
  showicon = expression {
    return cootx.isCreated(cooobj);
  }
  showunderline = expression {
    return cooobj.tlinvitationcategory == #MC_TeamRoom;
  }
}
```

For detailed information on the control parameters of the `FSCSIMLIST@1.1001:ListOption` control refer to [Faba10c]

11.4.26 ProcessOwnership

The customization point `ProcessOwnership` defines the owner and the group oft the workflow process. If no configuration was made the current user is used as the owner of the process and the group of the current role will be used as owning group oft he process.

The customization point `ProcessOwnership` is evaluated when creating a new process in the action `COOWF@1.1:CreateWorkFlowObject`.

Syntax

```
customize ProcessOwnership<objclass> {
   owner = expression {…}
   group = expression {…}
}
```

Description:

- *objclass*
  The customization applies to objects of the defined object class.

- *owner*
  Defines an expression to get the owner of the process.
- *group*
  Defines an expression to get the owning group of the process.

Example

app.ducx Customization Language

```
customize ProcessOwnership<Object> {
  owner = expression {
    TeamRoom tr = cooobj.objteamroom;
    if (tr && tr.HasClass(#TeamRoom)) {
      return tr.objowner;
    }
    else {
      return cooobj.objowner;
    }
  }
  group = expression {
    TeamRoom tr = cooobj.objteamroom;
    if (tr && tr.HasClass(#TeamRoom)) {
      return tr.objowngroup;
    }
    else {
      return cooobj.objowngroup;
    }
  }
}
```

12 app.ducx Expression Language

app.ducx Expression Language is a proprietary interpreted script language that allows you to access the Fabasoft Folio object model, and to invoke use cases. app.ducx Expressions can be embedded in `expression` blocks and other language elements of the domain-specific languages of Fabasoft app.ducx. This chapter provides an introduction to the app.ducx Expression Language.

Note: The grammar of the app.ducx Expression Language can be found in chapter 12.12 “Grammar of the app.ducx Expression Language”. The syntax for search queries is available in chapter 12.13 “Grammar of the app.ducx Query Language”.

12.1 General remarks concerning app.ducx expression language

app.ducx expression language is a distinct domain-specific language of Fabasoft app.ducx. app.ducx expressions can be embedded inline in an `expression` block in other domain-specific languages. However, it is also possible to create separate `.ducx-xp` files containing app.ducx expressions. App.ducx expression language files can be referenced from other domain-specific languages using the `file` keyword.

The app.ducx expression language is processed by the Fabasoft app.ducx compiler and transformed into Fabasoft app.ducx Expressions, which are evaluated at runtime by the Fabasoft Folio Kernel.

Keywords, predefined functions and predefined variables are not case sensitive.

12.1.1 Evaluating expressions at runtime

In the first step, the expression code is parsed. An expression can be parsed at runtime by calling the `Parse` method of the Fabasoft Folio Runtime. The `Parse` method returns an expression object

(which is not related to an object stored in the Fabasoft Folio). In a second step, the `Evaluate` method is invoked on the expression object for evaluating the expression. The scopes to be used during the evaluation of the expression must be passed to the `Evaluate` method. The result of the evaluation is passed back in the return value of the `Evaluate` method.

12.1.2 Testing expressions

The Fabasoft app.ducx Expression Tester (see next figure), allows you to test your expressions on the fly.

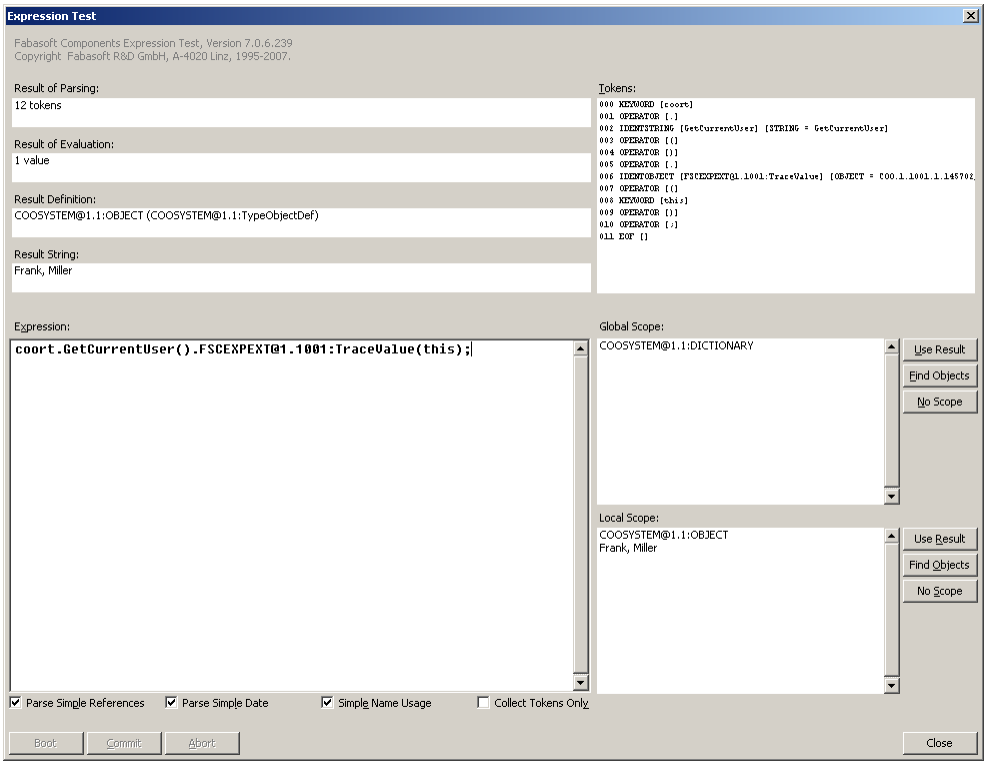


Figure 24: Fabasoft app.ducx Expression Tester

With the Fabasoft app.ducx Expression Tester, you can also set the contents of local and global scope to simulate the actual situation in which your expression will be evaluated.

12.1.3 Tracing in app.ducx expression language

Using app.ducx expression language, you can write trace messages to the Fabasoft app.ducx Tracer, which can be found in the `Setup\ComponentsBase\Trace` folder on your product DVD.

For writing messages and variable values to the trace output, you can either use the `%%TRACE` directive or the `Trace` method of the Fabasoft Folio Runtime.

If you pass two arguments to the `Trace` method of the Fabasoft Folio Runtime, the first argument is interpreted as a message while the second argument is treated as the actual value.

The `%%TRACE` directive can also be used to trace special objects like `cootx` to output all transaction variables defined for a transaction or `coometh` to output all set parameters within the implementation of a method.

Note:

- Values traced using the `%%TRACE` directive are only written to the Fabasoft app.ducx Tracer if trace mode is enabled for the corresponding software component whereas values traced using

the `Trace` method of the Fabasoft Folio Runtime are always written to the Fabasoft app.ducx Tracer.

- Keep in mind that the `%%TRACE` directive only works for method implementation and customization points.

For further information on enabling trace mode for a software component refer to chapter 13.1 “Tracing in Fabasoft app.ducx projects”.

Example

```
// Tracing string messages
coort.Trace("This message is written to the trace output");
// Tracing variable values
string mystrvar = "Hello world!";
object myobjvar = cooroot;
coort.Trace("Value of mystrvar", mystrvar);
coort.Trace("Value of myobjvar", myobjvar);
// Trace directives are only evaluated if the software component
// is in trace mode
%%TRACE("Value of mystrvar", mystrvar);
%%TRACE("Value of myobjvar", myobjvar);
// Tracing local and global scope
coort.Trace("Local variables", this);
coort.Trace("Global variables",::this);
```

12.2 Scopes

A scope is similar to a container holding a value that is accessible during the evaluation of an expression. The following distinct scopes are available to you when an expression is evaluated:

- local scope, which is accessed using the operator `:>`
- global scope, which is accessed using the operator `::`
- temporary scope, which is accessed using the operator `@`

You can use the keyword `this` along with the corresponding operator to access the value of a scope (e.g. `:>this` yields the value of the local scope, and `::this` the value of the global scope). However, in most cases the keyword `this` can be omitted. When accessing the local scope, you can also omit the operator `:>` in most cases.

Note: When using the selection operator `[]` for selecting values of lists or compound properties, the `:>` operator is required to access the local scope.

The keyword `declare` is used to declare an identifier. The Fabasoft app.ducx compiler automatically generates identifier declarations for use case parameters to allow access to parameters over the local scope when implementing a use case in app.ducx expression language.

For the following example, assume that the local scope `this` contains an instance of object class `APPDUCXSAMPLE@200.200:Order`, and that the temporary variable `@customer` contains an instance of object class `FSCFOLIO@1.1001:ContactPerson`. `APPDUCXSAMPLE@200.200:customerorders` is an object list pointing to instances of object class `APPDUCXSAMPLE@200.200:Order`. Within the square brackets put after `APPDUCXSAMPLE@200.200:customerorders`, `this` has a different meaning as it refers to each instance stored in the object list, and not to the local scope.

Example

```
ContactPerson @customer;
// Returns a STRINGLIST containing the names of all orders in property
// APPDUCXSAMPLE@200.200:customerorders
@customer.customerorders[objname];
// Returns an OBJECTLIST containing the orders whose name is identical to
```

```
// the name of the order in the local scope
@customer.customerorders[objname == :>objname];
```

In the following example, two strings, `isbn` and `title`, are declared in the local scope `this`. The temporary variable `@publication` is initialized with a compound structure consisting of two string properties, `isbn` and `title` that in turn are initialized using the two strings `isbn` and `title` from the local scope.

Note: Within the scope of the curly braces, `this` refers to the compound structure itself. To access the local scope, the `:>` operator must be used.

Example

```
string isbn = "000-0-00000-000-0";
string title = "An Introduction to Fabasoft app.ducx";
@publication = { isbn = :>isbn, title = :>title };
```

The temporary scope `@this` is only used for storing temporary values during the evaluation of an expression. Local scope `this` and global scope `::this` are similar to parameters, and can be populated with any valid Fabasoft Folio values when the `Evaluate` method is called for evaluating an expression.

The scopes cannot be changed while an expression is evaluated. However, you can add or modify dictionary entries if a scope contains a dictionary.

12.3 Types

For all variables within an expression a type should be defined explicitly.

Syntax

```
// Declaring a variable
Type variable;
// Declaring and initializing a variable
Type variable = initialValue;
```

Valid types are object classes and objects of the class `COOSYSTEM@1.1:TypeDefinition`, which includes basic data types (such as `COOSYSTEM@1.1:INTEGER`, `COOSYSTEM@1.1:STRING` or `COOSYSTEM@1.1:DATETIME`) as well as compound types and enumerations.

Shortcuts are provided for basic data types as defined by the object model language (see chapter 5.2.2.1.1 “Simple data types”) as well as Fabasoft Folio Kernel Interfaces as listed below.

Type	Description
`runtime`	The type for a Fabasoft Folio Runtime Interface, such as the predefined symbol `coort`.
`transaction`	The type for a Fabasoft Folio Transaction Interface, such as the predefined symbol `cootx`. An interface of this type is returned, if you create a new transaction using `coort.CreateTransaction()`.
`method`	The type for a Fabasoft Folio Method Interface, such as the predefined symbol `coometh`. An interface of this type is returned, if you obtain the implementation of a use case using `cooobj.GetMethod()`.

`searchresult`	The type for a Fabasoft Folio Interface to the result of an asynchronous search, as returned by `coort.SearchObjectsAsync()`.
`expression`	The type for a Fabasoft app.ducx Expression interface, as returned by `coort.Parse()`.
`aggregate`	The type for a Fabasoft Folio Aggregate Interface, which can be used as a generic substitute for any compound type.
`interface`	A generic type for a Fabasoft Folio Interface.

Table 34: Types for Fabasoft Folio Kernel Interfaces

Example

```
integer @bulksize = 150;
string @query = "SELECT objname FROM APPDUCXSAMPLE@200.200:Order";
searchresult @sr = coort.SearchObjectsAsync(cootx, @query);
Order[] @results = null;
while (count(@results = @sr.GetObjects(@bulksize)) > 0) {
  %%TRACE("Fetched chunk of search results", @results);
  if (@results != null) {
    @results.ProcessOrder();
  }
}
```

12.4 Operators

app.ducx expression language supports a wide range of operators.

12.4.1 Assignment operators

Assignment operators allow you to set property and variable values. The next table contains a list of supported assignment operators.

Operator	Description
`=`	The `=` operator is used for simple assignments. The value of the right operand is assigned to the left operand.
`+=`	Both operands are added, and the result is assigned to the left operand. The `+=` operator can be used with numeric data types, currencies and lists.
`-=`	The right operand is subtracted from the left operand, and the result is assigned to the left operand. The `-=` operator can be used with numeric data types, currencies and lists.
`*=`	Both operands are multiplied, and the result is assigned to the left operand. The `*=` operator can be used with numeric data types, currencies and lists.
`/=`	The left operand is divided by the right operand, and the result is assigned to the left operand. The `/=` operator can be used with numeric data types, currencies and lists.
`%=`	A modulus operation is carried out, and the result is assigned to the left operand. The `%=` operator can only be used with numeric data types, currencies and lists.

<<=	The <<= is used for character- and bitwise shifting to the left. The <<= operator can be used with strings, integers and currencies.
>>=	The >>= is used for character- and bitwise shifting to the right. The >>= operator can be used with strings, integers and currencies.

Table 35: Assignment operators

Example

```
Order @order;
Customer @customer;
// A simple assignment operation
@order.orderdate = coonow;
// Adding an element to a list
@customer.customerorders += @order;
// Adding an element to a list only if it is not part of the list already
@customer.customerorders *= @order;
```

12.4.2 Logical operators

Logical operators are implemented to support short circuit evaluation semantics. The right operand is only evaluated if the result of the evaluation is not determined by the left operand already. The next table shows a list of the supported logical operators.

Operator	Description
and (alternatively &&)	The and operator indicates whether both operands are true. If both operands have values of true, the result has the value true. Otherwise, the result has the value false. Both operands are implicitly converted to BOOLEAN and the result data type is BOOLEAN.
or (alternatively ||)	The or operator indicates whether either operand is true. If either operand has a value of true, the result has the value true. Otherwise, the result has the value false. Both operands are implicitly converted to BOOLEAN and the result data type is BOOLEAN.
not (alternatively !)	The expression yields the value true if the operand evaluates to false, and yields the value false if the operand evaluates to true. The operand is implicitly converted to BOOLEAN, and the data type of the result is BOOLEAN.

Table 36: Logical operators

Example

```
if (@orderstate == OrderState(OS_SHIPPED) and @orderdate != null or
  @orderstate == OrderState(OS_COMPLETED) and @invoice == null) {
  throw coort.SetError(#InvalidProcessingState, null);
}
```

12.4.3 Calculation operators

Table 37 contains a list of supported calculation operators.

Operator	Description

+ - * / %	The +, -, *, / and % operators are supported for numeric data types and lists. +, -, * and / are also supported for currencies (* and / need one integer or float operand). Additionally, the + operator can be used to concatenate strings.
++	The ++ increment operator is a unary operator that adds 1 to the value of a scalar numeric operand. The operand receives the result of the increment operation. You can put the ++ before or after the operand. If it appears before the operand, the operand is incremented. The incremented value is then used in the expression. If you put the ++ after the operand, the value of the operand is used in the expression before the operand is incremented.
--	The -- decrement operator is a unary operator that subtracts 1 from the value of a scalar numeric operand. The operand receives the result of the decrement operation. You can put the -- before or after the operand. If it appears before the operand, the operand is decremented. The decremented value is then used in the expression. If you put the -- after the operand, the value of the operand is used in the expression before the operand is decremented.
<<	The << is used for character- and bitwise shifting to the left. The << operator can be used with strings, integers and currencies. When used with lists, the right operand specifies the number of elements to be removed from the top of the list.
>>	The >> is used for character- and bitwise shifting to the right. The >> operator can be used with strings, integers and currencies. When used with lists, the right operand specifies the number of elements to be removed from the end of the list.

Table 37: Calculation operators

Example

```
// Check if every element of list @bbb is included in list @aaa
@aaa = ["John", "James", "Jim", "Jamie"];
@aaa = ["Jamie", "Jim"];
@aaa % @bbb == @aaa;
// Check if the last change of an object was carried out on the same date it was
// created. Using the "%86400" operation, the time portion of the datetime
// property "COOSYSTEM@1.1:objchangedat" is set to "00:00:00" in order to compare
// the date portion only using the "==" operator.
objcreatedat % 86400 == objchangedat % 86400
```

Note: If two different currencies are added or subtracted an implicit conversion is carried out. Following evaluation order is defined: The conversion table of the transaction variable TV_CURRCONVTAB is used. If not available, the conversion table of the left operand is used. If not available, the conversion table of the right operand is used. Otherwise an error is generated.

12.4.3.1 Examples for list operators

The +, -, *, / and % operators (concatenation, difference, union, symmetric difference, intersection) are supported for lists. The following example shows how list operators work.

Example

```
[]+[1] == [1];
```

```
[1,2,3]+[2,3,4] == [1,2,3,2,3,4];
[1,2,3]+[] == [1,2,3];

[1,2,3]-[2,3,4] == [1];
[1,2,3]-[1,2,3] == [];
[1,2,3]-[4,5,6] == [1,2,3];
[]-[1] == [];

[1,2,3]*[2,3,4] == [1,2,3,4];
[1,2,2]*[1,2,2,3,3,3,4] == [1,2,2,3,3,3,4];
[]*[1,2,2] == [1,2,2];
[1,2,3]*[] == [1,2,3];

[1,2,3]/[4,5,6] == [1,2,3,4,5,6];
[1,2,3]/[2,3,4] == [1,4];
[1,2,3]/[1,2,3] == [];

[1,2,3,4]%[2,3,4] == [2,3,4];
[1,2,3]%[2,3,4] == [2,3];
[1,2,3]%[4,5,6] == [];
[]%[4,5,6] == [];
```

12.4.3.2 Examples for dictionary operators

The `-`, `*`, `/` and `%` operators (difference, union, symmetric difference, intersection) are supported for dictionaries. These operators work on an element level, the value of a dictionary entry is not relevant. The left operand dominates. The following example shows how dictionary operators work.

Example

```
({})-({}) == ({});
({})-({ a:1, b:"x", c:true }) == ({});
({ a:1, b:"x", c:true })-({}) == ({ a:1, b:"x", c:true });
({ a:1, b:"x", c:true })-({ a:1, b:"x", c:true }) == ({});
({ a:1, b:"x", c:true })-({ b:2, c:"x", d:true }) == ({ a:1 });
({ a:1, b:"x", c:true })-({ a:2, b:"x", c:1 }) == ({});
({ a:2, b:"x", c:1 })-({ a:"x" }) == ({ b:"x", c:1 });

({})*({}) == ({});
({})*({ a:1, b:"x", c:true }) == ({ a:1, b:"x", c:true });
({ a:1, b:"x", c:true })*({}) == ({ a:1, b:"x", c:true });
({ a:1, b:"x", c:true })*({ a:1, b:"x", c:true }) == ({ a:1, b:"x", c:true });
({ a:1, b:"x", c:true })*({ b:2, c:"x", d:true }) == ({ a:1, b:"x", c:true, d:true });
({ a:1, b:"x", c:true })*({ a:2, b:"x", c:1 }) == ({ a:1, b:"x", c:true });
({ a:2, b:"x", c:1 })*({ a:"x" }) == ({ a:2, b:"x", c:1 });

({})/({}) == ({});
({})/({ a:1, b:"x", c:true }) == ({ a:1, b:"x", c:true });
({ a:1, b:"x", c:true })/({}) == ({ a:1, b:"x", c:true });
({ a:1, b:"x", c:true })/({ a:1, b:"x", c:true }) == ({});
({ a:1, b:"x", c:true })/({ b:2, c:"x", d:true }) == ({ a:1, d:true });
({ a:1, b:"x", c:true })/({ a:2, b:"x", c:1 }) == ({});
({ a:2, b:"x", c:1 })/({ a:"x" }) == ({ b:"x", c:1 });

({})%({}) == ({});
({})%({ a:1, b:"x", c:true }) == ({});
({ a:1, b:"x", c:true })%({}) == ({});
({ a:1, b:"x", c:true })%({ a:1, b:"x", c:true }) == ({ a:1, b:"x", c:true });
({ a:1, b:"x", c:true })%({ b:2, c:"x", d:true }) == ({ b:"x", c:true });
({ a:1, b:"x", c:true })%({ a:2, b:"x", c:1 }) == ({ a:1, b:"x", c:true });
({ a:2, b:"x", c:1 })%({ a:"x" }) == ({ a:2 });
```

12.4.4 Comparison operators

Comparison operators allow you to compare two operands. The next table provides a summary of the supported comparison operators. The data type of the result is always `BOOLEAN`.

Operator	Description
==	The equality operator compares two operands and indicates whether the

	value of the left operand is equal to the value of the right operand. The equality operator has a lower precedence than the relational operators (<, <=, >, >=).
!= (alternatively <>)	The inequality operator compares two operands and indicates whether the value of the left operand is not equal to the value of the right operand. The inequality operator has a lower precedence than the relational operators (<, <=, >, >=).
<	The relational operator < compares two operands and indicates whether the value of the left operand is less than the value of the right operand.
<=	The relational operator <= compares two operands and indicates whether the value of the left operand is less than or equal to the value of the right operand.
>	The relational operator > compares two operands and indicates whether the value of the left operand is greater than the value of the right operand.
>=	The relational operator >= compares two operands and indicates whether the value of the left operand is greater than or equal to the value of the right operand.
<=>	The relational operator <=> compares two operands. It returns -1 if the left value is lower and +1 if the left value is greater than the right value. If the values are equal, the result is 0.
`contains`	The `contains` operator determines whether left operand contains the right operand. This operator can be used with string operands. It may be preceded by the `not` keyword.
`like`	The `like` operator determines whether the left string matches the right string. The `%` and _ wildcards can be used in the right string operand. The `like` operator can be preceded by the `sounds` keyword for a phonetic comparison. Furthermore, it can also be preceded by the `not` keyword.
`in`	The `in` operator determines whether the value of the left operand is an element of the list provided in the right operand. The `in` operator can also be used with a list in the left operand. It may be preceded by the `not` keyword.
`includes`	The `includes` operator determines whether the value of the right operand is an element of the list provided in the left operand. It may be preceded by the `not` keyword.
`between and`	The `between and` operator determines whether the value of the left operand is in the range between the values of the operands provided after the keywords `between` and `and`.
`is null`	The `is null` operator returns `true` if the value of the left operand is undefined.

Table 38: Comparison operators

Example

```
if (@points < 100) {
  @memberstatus = "MS_SILVER";
}
else if (@points between 100 and 1000) {
  @memberstatus = "MS_GOLD";
}
else {
  @memberstatus = "MS_PLATINUM";
}
if (@status in ["MS_GOLD", "MS_PLATINUM"]) {
  @expressshipping = true;
}
if (@nickname like "Bob%" OR @nickname in ["Dick", "Rob"]) {
  @firstname = "Robert";
}
```

Note: When comparing aggregates or dictionaries, the values of all attributes or entries are compared. The comparison is recursive for nested values.

Note: Aggregate types can specify a comparator method in `COOSYSTEM@1.1:typecompare`. When comparing such aggregates this method is used to calculate the result of the comparison. Standard comparison of aggregates only allow a check for equality of values (operators `==`, `!=` , `in`, and `includes`). Using `COOSYSTEM@1.1:typecompare` allows implementation of greater or less operators for aggregates.

Note: Objects and dictionaries only allow a check for equality of values (operators `==`, `!=` , `in`, and `includes`).

Note: Contents cannot be compared by their value.

Note: String operands are compared using the setting `COOSYSTEM@1.1:domaincisqry` in your current domain object. The default for this setting is true, meaning that comparison is case insensitive by default. This is also relevant for `min`/`max`/`sort`/`unique`/`find` and the list operators `-`, `*`, `/`, and `%`.

12.4.5 Conditional operator

The conditional operator `?:` has three operands. It tests the result of the first operand, and then evaluates one of the other two operands based on the result of the evaluation of the first operand. If the evaluation of the first operand yields `true`, the second operand is evaluated. Otherwise, the third operand is evaluated.

Example

```
@orders = (@customer != null) ?
  @customer.customerorders[objname] :
  null;
```

12.4.6 Selection operator

The selection operator `[]` can be used for the following purposes:

- as a list constructor to define a list of values,
- to select elements from a list of values
- to filter elements of a list,
- to specify a parameter as the return value when invoking a use case, and
- for calculated identifiers (see chapter 12.8 “Calculated identifiers”).

Example

```
// Constructing an empty list
@productcategories = [];
// Constructing a list of string values
@productcategories = ["Fish", "Meat", "Poultry"];
// Selecting elements from a list
// The result is a single element
@fish = @productcategories[0];
@meat = @productcategories[1];
// Selecting multiple elements from a list
// The result is a list again
@nofish = @productcategories[1,2];
// Selecting elements starting from the end of the list by specifying negative indices
@poultry = @productcategories[-1];
// Example for filtering a list: This expression returns the orders that
// do not have a valid order date.
// The result is again a list if more than one item is selected
@customer.customerorders[orderdate is null];
// Specifying a parameter as the return value when invoking a use case
@neworder = #Order.ObjectCreate()[2];
// Results in the method object of the call
@neworder = #Order.ObjectCreate()[...];
```

12.4.7 $ operator

By default, the Fabasoft Folio Kernel tries to interpret identifiers as references when evaluating expressions. In order to use an identifier that could also be interpreted as a reference as a variable, it must be prefixed with `$`.

Please note that the Fabasoft app.ducx compiler will attempt to automatically do that for you if it can determine the context in which you use the identifier, if it can't you will receive a warning.

For the following example, assume that the local scope contains a dictionary. `objname` can only be used as a variable when prefixed with `$`.

Example

```
// Assuming the local scope contains a dictionary:
$objname = "Hello world";
// When omitting the "$", the expression is interpreted as follows:
this.COOSYSTEM@1.1:objname = "Hello world";
```

12.4.8 # operator

If you need to retrieve a component object in an expression, you must prefix its fully qualified reference with `#`. For instance, when referring to a property definition or an object class, you must prefix the reference with `#` in order to get the corresponding component object.

Example

```
// Accessing property definition COOSYSTEM@1.1:objname
@objnameprop = #objname;
@ordername = @order.GetAttributeValue(cootx, @objnameprop);
// Accessing object class APPDUCXSAMPLE@200.200:Order
@orderclass = #APPDUCXSAMPLE@200.200:Order;
@neworder = @orderclass.ObjectCreate()[2];
```

12.5 Predefined variables and functions

app.ducx expression language provides you with a set of predefined variables and functions to make programming as convenient as possible.

12.5.1 Predefined variables

Table 39 shows the list of predefined variables that are provided automatically by the Fabasoft Folio Kernel when an expression is evaluated.

Variable	Description
coort	The coort variable can be used to access the Fabasoft Folio Runtime.
cootx	The current transaction context is accessible by the cootx variable.
cooobj	The cooobj variable holds the current object on which the evaluation of the expression is invoked.
coometh	For use case implementations in app.ducx expression language, the coometh variable holds the so-called method context.
coouser	The coouser variable holds the user object of the current user.
cooenv	The cooenv variable holds the user environment of the current user.
cooroot	The cooroot variable holds the desk of the current user.
coolang	The coolang variable holds the language of the current user.
coodomain	The coodomain variable holds the current domain.
coonow	The coonow variable holds the current date and time.

Table 39: Predefined variables

Example

```
// Using the Fabasoft Folio Runtime
@user = coort.GetCurrentUser(); // or coouser
@desk = coort.GetCurrentUserRoot();// or cooroot
@currentdate = coonow;

// Accessing transaction variables using the generic interface
cootx.SetVariableValue(#APPDUCXSAMPLE@200.200, 1, #BOOLEAN, 0, true);
@txvarval = cootx.GetVariableValue(#APPDUCXSAMPLE@200.200, 1);

// Using the current object
@name = cooobj.GetName(cootx);
@objcls = cooobj.GetClass();
@orders = cooobj.APPDUCXSAMPLE@200.200:customerorders;

// Using the method context to call the super method
cooobj.CallMethod(cootx, coometh);
```

12.5.2 Implicit properties of the STRING data type

Values of data type STRING currently only support the implicit property length, which returns the length of the string in characters.

Example

```
@mystr = "Hello world!";
@mystrlen = @mystr.length;
```

12.5.3 Implicit properties of the DATETIME data type

Values of data type `DATETIME` expose implicit properties that can be used to access the individual date and time components. These implicit properties are listed in the next table. All of these implicit properties are of data type `INTEGER`.

Property	Description
`year`	The `year` property can be used for getting and setting the year.
`month`	The `month` property can be used for getting and setting the month.
`day`	The `day` property can be used for getting and setting the day.
`hour`	The `hour` property can be used for getting and setting the hour.
`minute`	The `minute` property can be used for getting and setting the minutes.
`second`	The `second` property can be used for getting and setting the seconds.
`dayinweek`	The `dayinweek` property can be used for getting the day of the week, with the value `0` representing Sunday and `6` representing Saturday.
`dayinyear`	The `dayinyear` property can be used for getting the day of the year, with the possible values ranging from `1` to `366` (for a leap year).
`weekinyear`	The `weekinyear` property can be used for getting the week of the year, with the possible values ranging from `1` to `52`.
`local`	The `local` property returns the date and time value converted to local time.
`universal`	The `universal` property returns the date and time value converted to universal time.

Table 40: Implicit DATETIME properties

Example

```
@currentdate = coonow;
@currentdate.hour = 12;
@currentdate.minute = 0;
@currentdate.second = 0;
@closed = @currentdate.dayinweek == 0;
```

12.5.4 Implicit pseudo functions

Table 41 contains a list of implicit pseudo functions supported by app.ducx expression language. These functions can be invoked on an object.

Function	Description

`IsClass(class)`	`IsClass` determines whether the object class it is invoked on is derived from or identical to `class`.
`HasClass(class)`	`HasClass` determines whether the object is an instance of or derived from `class`.
`GetClass()`	`GetClass` returns the object class of the object.
`GetName(tx)`	`GetName` returns the *Name* (`COOSYSTEM@1.1:objname`) of the object.
`GetAddress()`	`GetAddress` returns the *Address* (`COOSYSTEM@1.1:objaddress`) of the object.
`GetReference()`	`GetReference` returns the *Reference* (`COOSYSTEM@1.1:reference`) of a component object.
`GetIdentification()`	`GetIdentification` returns the full identification of the object, which is a combination of the *Address* (`COOSYSTEM@1.1:objaddress`) and a version timestamp.

Table 41: Implicit pseudo functions

Example

```
@objectclass = cooobj.GetClass();
@objectname = cooobj.GetName(cootx);
@objectaddress = cooobj.GetAddress();

cooobj.HasClass(#APPDUCXSAMPLE@200.200:Order) ?
  cooobj.APPDUCXSAMPLE@200.200:ProcessOrder() : null;
```

12.5.5 Working with contents and dictionaries

Please refer to chapter 8.4.8.3 “Working with contents” for a list of methods that can be invoked on variables of type `CONTENT`, and to chapter 8.4.8.4 “Working with dictionaries” for a list of methods that can be invoked on variables of type `DICTIONARY`.

Example

```
// Assuming that the global scope contains a dictionary
if (::HasEntry("description")) {
  // Initialize a new content
  content @description = {};
  // Set the string contained in ::description into the content
  // (use 65001 as "codepage" for UTF-8 encoding)
  @description.SetContent(cootx, 1, 65001, ::description);
  // Remove the description string from the global scope dictionary
  ::ClearEntry("description");
}
```

12.5.6 Getting the data type of an expression

The `typeof` function allows you to determine the data type of an expression.

Example

```
// Determining the data type of the local scope
@localtype = typeof(this);

// Determining the data type of the global scope
```

```
@globaltype = typeof(::this);
// Determining the data type of a variable
@myvalue = "Hello world!";
@resulttype = typeof(@myvalue);
// Determining the data type of an expression
@resulttype = typeof(cooobj.APPDUCXSAMPLE@200.200:customerorders);
```

12.5.7 String functions

Table 42 contains the list of string utility functions.

Function	Description
`upper(string)`	The `upper` function converts all characters of a string to upper case.
`lower(string)`	The `lower` function converts all characters of a string to lower case.
`indexof(string, pattern)`	The `indexof` function returns the character-based index of `pattern` within `string`. If the pattern in not found the function returns -1
`strlen(string)`	The `strlen` function returns the length of `string`.
`strtrim(string)`	The `strtrim` function trims white space at the beginning and at the end of `string`.
`strhead(string, index)`	The `strhead` function extracts the leftmost `index` characters from a string and returns the extracted substring. `index` is zero-based.
`strtail(string, index)`	The `strtail` function extracts the characters from a string starting at the position specified by `index` and returns the extracted substring. `index` is zero-based.
`strsplit(string, separator)`	The `strsplit` function identifies the substrings in `string` that are delimited by `separator`, and returns a list containing the individual substrings.
`strjoin(list [, separator])`	The `strjoin` function concatenates the `list` of strings and inserts `separator` between the individual elements yielding a single concatenated string. If `separator` is not specified or `null` then the list elements are concatenated directly.
`strreplace(string, from [, to])`	The `strreplace` function replaces all occurrences of string `from` with string `to` in `string`. If `to` is not specified or null `null` then all occurences of `from` are deleted from the string.

Table 42: String functions

Example

```
@value = strhead("ABC", 1);                    // yields "A"
@value = strtail("ABC", 1);                    // yields "BC"
@value = strsplit("ABC", "B");                 // yields ["A","C"]
@value = strjoin(["A", "B", "C"], "");         // yields "ABC"
@value = strjoin(strsplit("A-B-C", "-"), "+"); // yields "A+B+C"
@value = strreplace("ABC", "B", "");           // yields "AC"
@value = strreplace("ABCABC", "B", "X");       // yields "AXCAXC"
@value = strlen("ABC");                        // yields 3
```

In addition to the string functions provided by app.ducx expression language, the actions listed in the next table are useful for manipulating strings. For further information, refer to the Fabasoft reference documentation (see chapter 3.7.30 “Fabasoft Reference Documentation”).

Function	Description
`COOSYSTEM@1.1:Format(value, pattern, symbols, result)`	This action takes a single value (any type) as first parameter and a formatting pattern as second parameter. The third parameter is for advanced options (code page, custom symbols for separators or the decimal point). The result is returned in the fourth parameter. Refer to the Fabasoft reference documentation for a description of the supported formatting patterns.
`COOSYSTEM@1.1:Print(string, ...)`	Processes a format string or prints the object to a resulting string. If the `string` parameter contains a non-empty format string, this is used regardless of the object of the action. If the object is a string object, the property `Print COOSYSTEM@1.1:string` is used as format string. If the object is an error message, the property `COOSYSTEM@1.1:errtext` is used as format string. In all other cases the name of the object is used as format string. If the string contains formatting patterns starting with the "%" character these patterns are replaced by the additional parameters of the `Print` action. Refer to the Fabasoft reference documentation for a description of the supported formatting patterns.
`COOSYSTEM@1.1:PrintEx(string, arguments)`	Uses `COOSYSTEM@1.1:Print` to print a format string or an object to a resulting string. Parameters for formatting are passed in a string list in the parameter `arguments`. Each line in the string list in the `arguments` parameter is evaluated as an expression.

Table 43: Actions for manipulating strings

Example

```
// Get the modification date of the object as formatted string
string @formattedstr = cooobj.Format(cooobj.objmodifiedat, "dT");
// Format an integer value with leading zeroes to yield "000123"
integer @intval = 123;
string @intvalstr = cooobj.Format(@intval, "000000");
// Format an integer value using digit grouping to yield "123,456"
integer @intval2 = 123456;
string @intvalstr2 = cooobj.Format(@intval2, "#,###");
// Get the object's name and subject as formatted string
string @titlestr = cooobj.Print("%s (%s)", cooobj.objname, cooobj.objsubject);
// Assuming that StrCustomFormat is a string object containing the format pattern
// "%s iteration %03d", the following expression will write the name of the current object
```

```
// and the current loop iteration number padded with leading zeroes to the tracer
for (integer @idx = 0; @idx < 1000; @idx++) {
  %%TRACE(#StrCustomFormat.Print(null, cooobj.objname, @idx));
}
```

12.5.8 List functions

The next table shows the list of functions provided for working with lists.

Function	Description
`count(list)`	The `count` function returns the number of elements in `list`.
`insert(list, index, value)`	The `insert` function inserts `value` into `list` at position `index`. `index` is zero-based. If `index` is greater than the number of elements in `list`, `value` is appended at the end of `list`.
`delete(list, index [, count])`	The `delete` function deletes the value at `index` from `list`. `index` is zero-based. If `count` is not specified or `null` then one element is deleted, otherwise `count` specifies the number of elements following `index` that should be deleted. If less than `count` elements are available, the list is truncated at `index`. If `count` is negative that elements before `index` are deleted.
`find(list, value)`	The `find` function searches `list` for the element `value`, and returns the index of the first occurrence within the entire list. If `value` is not found in `list`, the number of elements in `list` is returned.
`sort(list)`	The `sort` function sorts the elements in `list`.
`unique(list)`	The `unique` function makes the elements in `list` unique.
`revert(list)`	The `revert` function reverts the elements in `list`.

Table 44: List functions

Example

```
insert(@orders, count(@orders), @neworder);
delete(@orders, find(@orders, @canceledorder));
unique(sort(@orders));
```

12.5.9 Mathematical functions

The next table shows the list of mathematical functions supported by app.ducx expression language.

Function	Description
`sum(list)`	The `sum` function returns the sum of all values in `list`. Values can also be passed to `sum` as individual arguments. The `SUM` function can only be used with numeric data types.
`avg(list)`	The `avg` function returns the average of all values in `list`. Values can also be passed to `avg` as individual arguments. The `avg` function can only be used with numeric data types.

`min(list)`	The `min` function returns the smallest value in `list`. Values can also be passed to `min` as individual arguments. The `min` function can be used with strings and numeric data types.
`max(list)`	The `max` function returns the largest value in `list`. Values can also be passed to `max` as individual arguments. The `max` function can be used with strings and numeric data types.

Table 45: Mathematical functions

Example

```
@avgorderamount = avg(@customer.APPDUCXSAMPLE@200.200:customerorders.
  APPDUCXSAMPLE@200.200:ordertotal.currvalue);
```

12.5.10 Escape sequences for special characters

Table 46 contains a list of supported escape sequences for special characters.

Character	ASCII representation	ASCII value	Escape sequence
New line	`NL (LF)`	`10` or `0x0a`	`\n`
Horizontal tab	`HT`	`9`	`\t`
Vertical tab	`VT`	`11` or `0x0b`	`\v`
Backspace	`BS`	`8`	`\b`
Carriage return	`CR`	`13` or `0x0D`	`\r`
Form feed	`FF`	`12` or `0x0C`	`\f`
Alert	`BEL`	`7`	`\a`
Backslash	`\`	`92` or `0x5C`	`\\`
Question mark	`?`	`63` or `0x3F`	`\?`
Single quotation mark	`'`	`39` or `0x27`	`\'`
Double quotation mark	`"`	`34` or `0x22`	`\"`
Octal number	`ooo`		`\ooo`
Hexadecimal number	`hhh`		`\xhhh`
Null character	`NUL`	`0`	`\0`

Table 46: Escape sequences for special characters

Example

```
@fullname = "Samuel \"Sam\" Adams";
```

12.6 Getting and setting property values

Syntax

```
// Getting property values using the assignment operator
variable = object.property;
// Setting property values using the assignment operator
object.property = expression;
// Setting values of a compound property list using the assignment operator
// The values are assigned based on their position
object.property = [{val1, val2, ...}, {vala, valb, ...}, ...];
// Simple initialization of compound properties
CompoundType compoundtype = { val1, val2, ... };
CompoundType({ val1, val2, ... });
// Setting values of a compound property list using the assignment operator
// The values are assigned based on their position
object.property = [{val1, val2, ...}, {vala, valb, ...}, ...];
// Setting values of a compound property list using the assignment operator
// The values are assigned based on the properties of the compound type
object.property = [{reference1 = val1, reference2 = val2, ...}, ...];


// Getting property values using the Get... functions
variable = object.GetAttributeValue(transaction, property);
variable = object.GetAttribute(transaction, property);
variable = object.GetAttributeString(transaction, property, language);
variable = object.GetAttributeStringEx(transaction, property, language,
  additionallist, displayflags);
// Setting property values using the Set... functions
object.SetAttributeValue(transaction, property, index, value);
object.SetAttribute(transaction, property, value);
```

Note: You can set the value (`currvalue`) of a currency property (compound type) by just assigning a string, float or integer to the currency property (`currency @cur = 7.56;`). In this case it is not necessary to specify explicitly the `currvalue` property (`@cur.currvalue = 7.56;`).

You can use the assignment operator `=` to get and set the value of a property. Even though using the assignment operator is recommended, you may also use the following functions to get and set property values:

- The `GetAttributeValue` function is used for retrieving a scalar property value.
- The `GetAttribute` function is used for retrieving a list.
- The `GetAttributeString` function is used for retrieving a string representation of the property value. This function can be used for retrieving multilingual strings in the desired language.
- The `GetAttributeStringEx` function is used for retrieving a string representation of the property value. This function allows you to specify so-called display flags (see next table) for the value to be retrieved.
- The `SetAttributeValue` function is used to store a scalar value in a property.
- The `SetAttribute` function is used to store a list.

Value	Description
`0x0001`	Object address

0x0002	Object reference
0x0004	Enumeration reference
0x0010	XML escaped
0x0020	HTML escaped
0x1000	Row list
0x2000	Column list
0x8000	No list indicator

Table 47: Display flags for GetAttributeStringEx

Example

```
@orders = @customer.APPDUCXSAMPLE@200.200:customerorders;
@orders = @customer.GetAttribute(cootx, #APPDUCXSAMPLE@200.200:
  customerorders);
@orderdate = @order.GetAttributeValue(cootx, #APPDUCXSAMPLE@200.200:
  orderdate);
@order.APPDUCXSAMPLE@200.200:orderdate = coonow;
@order.SetAttributeValue(cootx, #APPDUCXSAMPLE@200.200:orderdate, 0,
  coonow);
@engname = @product.GetAttributeString(cootx, #mlname, #LANG_ENGLISH);
@gername = @product.GetAttributeString(cootx, #mlname, #LANG_GERMAN);
@description = @product.GetAttributeStringEx(cootx, #APPDUCXSAMPLE@200.200:
  productdescription, null, null, 0x1000);
```

12.7 Invoking use cases

Syntax

```
object.usecase(parameter, ...)
```

A use case can only be invoked on an object. The full reference of the use case to be invoked must be provided, followed by the list of parameters, which must be enclosed in parentheses. Multiple parameters must be separated by commas.

You do not need to specify optional parameters. They can either be omitted in the parameter list, or denoted as `null`.

There are three methods for retrieving output parameters in app.ducx expression language:

- If you need to retrieve only one output parameter, the selection operator `[]` specifying the result parameter position can be used.
- If you need to retrieve the method object of the call, the selection operator `[...]` can be used.
- If you need to retrieve multiple output parameters, variables must be specified for the output parameters in the parameter list and prefixed with an ampersand (`&`).

Example

```
// Optional parameters can be omitted
@order.COODESK@1.1:ShareObject(, , #APPDUCXSAMPLE@200.200:customerorders,
  @customer);
```

```
// "null" can also be passed in for optional parameters
@order.COODESK@1.1:ShareObject(null, null, #APPDUCXSAMPLE@200.200:
  customerorders, @customer);
// Retrieving an output parameter: method 1 - specify the return value
@neworder = #APPDUCXSAMPLE@200.200:Order.ObjectCreate()[2];
// Retrieving an output parameter: method 2
method @meth = #APPDUCXSAMPLE@200.200:Order.ObjectCreate()[...];
@neworder = @meth.GetParamteter(2);
// Retrieving an output parameter: method 3
#APPDUCXSAMPLE@200.200:Order.ObjectCreate(null, &@neworder);
```

12.8 Calculated identifiers

Syntax

```
// Accessing a property using a calculated identifier
object.[expression]
// Invoking a use case using a calculated identifier
object.[expression](parameter, ...)
```

The selection operator `[]` can be used to specify an expression yielding a calculated identifier for accessing a property or invoking a use case.

For a calculated identifier, the expression specified in square brackets is evaluated, and then the result is interpreted as a property definition, an action or a use case.

Example

```
// Assigning a value to a calculated property
@attrdef = #APPDUCXSAMPLE@200.200:orderdate;
@order.[@attrdef] = coonow;
// Invoking a use case using a calculated identifier
@createinvoice = #APPDUCXSAMPLE@200.200:CreateInvoice;
@customer.[@createinvoice](@orders, &@invoice);
```

12.9 Accessing the transaction context

The `cootx` variable can be used to access the current transaction context. The most important methods of a transaction are listed in chapter 8.4.10 “Accessing the transaction context”.

In some scenarios it is necessary to carry out operations in a separate transaction. Any changes that have been made in a new transaction can be committed or rolled back separately from the main transaction.

Example

```
usecase CreateInvoice() {
  variant Order {
    impl = expression {
      // Create a new transaction
      interface @extension = coort.GetExtension();
      transaction @localtx = @extension.CreateTransaction();
      transaction @backuptx = cootx;
      try {
        coort.SetThreadTransaction(@localtx);
        Invoice @invoice = #Invoice.ObjectCreate();
        @invoice.APPDUCXSAMPLE_200_300_InitializeInvoice();
        @localtx.Commit();
      }
      catch (error) {
        @localtx.Abort();
```

```
      }
      finally {
        // Restore original transaction context
        coort.SetThreadTransaction(@backuptx);
        // Clear the variables holding the transactions
        @backuptx = null;
        @localtx = null;
      }
    }
  }
}
```

Note: A better and simpler way to create transactions is using the `try new transaction` statement. Please refer to chapter 12.10.5 “Creating new transactions or opening a transaction scope” for further information.

12.9.1 Working with transaction variables

Syntax

```
// Retrieving the value stored in a transaction variable
value = #TV.reference;
// Storing a value in a transaction variable
#TV.reference = value;
```

The `#TV` object is a special object that provides access to transaction variables.

Note: Transaction variables can also be accessed using the `cootx` variable. Please refer to chapter 12.5.1 “Predefined variables” for further information.

Example

```
// Retrieving the value stored in a transaction variable
@printinvoice = #TV.TV_PRINTINVOICE;
// Storing a value in a transaction variable
#TV.TV_INVOICE = cooobj;
```

12.10 Structured programming with expressions

For your convenience, app.ducx expression language supports all common language constructs that you already know from most imperative programming languages.

12.10.1 Conditions

Syntax

```
if (expression) {
  ...
}
else if (expression) {
  ...
}
else {
  ...
}

switch (expression) {
  case constant:
    ...
    break;
  case constant:
```

```
    ...
    break;
  default:
    ...
}
```

You can use `if` statements in app.ducx expression language. The `if` keyword must be followed by parentheses enclosing a conditional expression, and non-optional curly braces. An `if` block can be followed by multiple `else if` blocks and an optional `else` block.

Example

```
@orderstate = @order.orderstate;
if (@orderstate == OrderState(OS_PENDING)) {
  @order.ProcessPendingOrder();
}
else if (@orderstate == OrderState(OS_SHIPPED)) {
  @order.ProcessShippedOrder();
}
else {
  throw coort.SetError(#OrderAlreadyProcessed, null);
}
// lists as conditional expression are evaluated true, if the list contains at least
// one not null element
if (["", "", "a"]) {
  true;
}
```

Note: It is not necessary that `OS_PENDING` is explicitly casted (e.g. `@orderstate == OS_PENDING` works, too).

The `switch` - `case` - `default` statement can be used to evaluate the switch expression and execute the appropriate case.

Example

```
OrderState @orderstate;
switch (@orderstate){
  case OS_PENDING:
    @state = 1;
    break;
  case OS_SHIPPED:
    @state = 2;
    break;
  default:
    @state = 0;
}
```

Note: Enumeration items like `OS_PENDING` are determined by `@orderstate` in the `switch` statement.

12.10.2 Loops

Syntax

```
for (expression) {
  ...
}
while (expression) {
  ...
}
do {
```

```
  ...
} while (expression);
```

app.ducx expression language allows you to define loops with both a constant and a non-constant number of iterations. The common statement constructs of imperative programming languages are supported: `for`, `while`, and `do-while` loops. The blocks denoted by curly braces are required. The `break` statement can be used to exit a loop. The `continue` statement can be used to skip the remainder of the loop body and continue with the next iteration of the loop.

Example

```
float @totalvalue;
for (integer @idx = 0; @idx < count(orderpositions); @idx++) {
  Product @product = @positions[@idx].product;
  @totalvalue += FLOAT(@product.unitprice.currvalue) *
    @positions[@idx].quantity;
}
for (OrderPosition @position : orderpositions) {
  Product @product = @position.product;
  if (@product != null) {
    currency @total += @product.unitprice * @position.quantity;
  }
}
integer @stock = @product.itemsinstock;
integer @threshold = @product.productionthreshold;
while (@stock <= @threshold) {
  @product.ProduceItem();
  @stock++;
}
do {
  @order.ProcessOrder(&@orderstate);
} while (@orderstate != OrderState(OS_COMPLETED));
```

12.10.3 Raising an error

Syntax

```
// Raising a custom error
throw coort.SetError(errormessage, argument);
// Rethrowing an exception
throw errorcode;
```

Using the `throw` keyword, app.ducx expression language allows you to raise an error:

- If you want to rethrow an exception in an error handler (e.g. in a `catch` block), you just need to specify the error code of the exception after the `throw` keyword.
- If you want to raise a custom error, you need to issue a call to the `SetError` function of the Fabasoft Folio Runtime.
- Instead of using the `SetError` function you can also use the `COOSYSTEM@1.1:RaiseError` action, which allows you to pass in additional arguments for filling a formatting string. The `throw` keyword is not required when using `COOSYSTEM@1.1:RaiseError`.

Example

```
// Raising a custom error using a "throw" statement
throw coort.SetError(#InvalidInvoice, null);
// Raising a custom error using the COOSYSTEM@1.1:RaiseError action, assuming that the
// error message APPDUCXSAMPLE@200.200:InvalidInvoice contains a formatting pattern like
// "Invoice '%s' (no. %d) is not valid!"
```

```
cooobj.RaiseError(#InvalidInvoice, cooobj.objname, cooobj.invoicenumber);
```

12.10.4 Error handling

Syntax

```
try {
  ...
}
catch (condition) {
  ...
}
finally {
  ...
}
```

app.ducx expression language supports `try-catch-finally` statement constructs for handling exceptions. A `try` block must be followed by at least one `catch` block, followed by an optional `finally` block.

If an exception occurs when processing the `try` block, the Fabasoft Folio Kernel tries to locate a `catch` block with a condition matching the error code of the exception.

A `catch` block can have three distinct types of conditions:

- The error code to be handled can be specified in form of an integer number. In this case, the corresponding `catch` block is only executed, if the error code specified matches the error code of the exception.
- A variable can be specified. If an exception occurs, the error code is stored in the specified variable, and the corresponding `catch` block is executed. The variable can be used for reading the error code and the corresponding error message.
- The `...` operator can be specified to handle all exceptions without taking into account the error code.

The optional `finally` block is executed after the `try` and `catch` blocks have been processed, whether an exception occurred or not.

Note: No exception is raised if an error occurs during the evaluation of the condition of a conditional statement. In such a case, the `else` block – if present – is executed instead.

Example

```
try {
  // Exceptions that might be thrown by APPDUCXSAMPLE@200.200:ProcessOrder
  // are handled in the catch block
  @order.APPDUCXSAMPLE@200.200:ProcessOrder();
}
catch (@error) {
  // Write error message to trace output and rethrow the exception
  coort.Trace(coort.GetErrorText(@error));
  throw @error;
}
finally {
  // Do some cleanup; this block is executed in any case
  @order.APPDUCXSAMPLE@200.200:CleanUp();
}
if (10 / 0) {
  // This block is never executed
}
else {
  // This block is always executed because the division by zero in the
  // condition of the "if" statement causes an error
```

```
}
```

12.10.5 Creating new transactions or opening a transaction scope

Similar to the `try` statement for error handling it is possible to execute a block using a separate transaction context:

Example

```
try new transaction {
  // The statements in this block are executed in a new transaction context.
  Invoice @invoice = #Invoice.ObjectCreate();
  @invoice.APPDUCXSAMPLE_200_300_InitializeInvoice();
}
catch (...) {
  // Here you can catch exceptions that have occurred during the try block
}
finally {
  // Here you can explicitly commit or abort the transaction context
  // using cootx.Commit() or cootx.Abort().
  // If the finally block is omitted the transaction context is automatically
  // committed if no exception has occurred.
  // Otherwise the transaction context is aborted.
}
```

If the `new` keyword is omitted, a transaction scope is opened. A transaction scope is a sub transaction of the current transaction. When a transaction scope is commited, the changes of that scope are propagated to the surrounding transaction. These changes are only persisted if the surrounding transaction context is commited.

Example

```
try transaction {
  // The statements in this block are executed in a new transaction scope.
  Invoice @invoice = #Invoice.ObjectCreate();
  @invoice.APPDUCXSAMPLE_200_300_InitializeInvoice();
}
catch (...) {
  // Here you can catch exceptions that have occurred during the try block
}
finally {
  // Here you can explicitly commit or abort the transaction scope
  // using cootx.Commit() or cootx.Abort().
  // If the finally block is omitted the transaction scope is automatically
  // committed to the surrounding transaction context if no exception has occurred.
  // Otherwise the transaction scope is aborted.
}
```

12.10.6 Returning values

Syntax

```
return expression;
```

The `return` statement can be used to stop the evaluation of an expression at any time. Each expression has a return value, which is calculated by the expression following the `return` keyword.

Example

```
if (@order != null) {
  return @order;
```

```
}
```

12.10.7 Directives

A directive is a special statement that does not influence the semantic of the expression.

Syntax

```
%%NAME(parameters);
```

12.10.7.1 %%TRACE

The `%%TRACE` directive can be used to conditionally write trace messages to the Fabasoft app.ducx Tracer (see chapter 13.1 “Tracing in Fabasoft app.ducx projects”).

Syntax

```
%%TRACE(message);
%%TRACE(value);
%%TRACE(message, value);
```

Example

```
%%TRACE("Hello World!");
%%TRACE(cooobj);
%%TRACE("Current Object", cooobj);
%%TRACE(cooobj.objname + " locked?", cooobj.objlock.objlocked);
```

12.10.7.2 %%FAIL

The `%%FAIL` directive can be used to generate a failure. The message is written to the Fabasoft app.ducx Tracer and an error (`EXPRERR_FAIL`) is raised.

Note: Like the `%%TRACE` directive, the `%%FAIL` directive is only evaluated if trace mode is activated for your software component (see chapter 13.1 “Tracing in Fabasoft app.ducx projects”).

Syntax

```
%%FAIL;
%%FAIL(message);
```

Example

```
%%FAIL;
%%FAIL("Unexpected!");
```

12.10.7.3 %%ASSERT

The `%%ASSERT` directive can be used to check conditions. In case the condition returns `false`, a message is written to the Fabasoft app.ducx Tracer and an error (`EXPRERR_ASSERT`) is raised.

Note: Like the `%%TRACE` directive, the `%%ASSERT` directive is only evaluated if trace mode is activated for your software component (see chapter 13.1 “Tracing in Fabasoft app.ducx projects”).

Syntax

```
%%ASSERT(condition);
%%ASSERT(message, condition);
%%ASSERT(message, expectedvalue, actualvalue);
```

Example

```
%%ASSERT(cooobj.objlock.objlocked);
%%ASSERT("'cooobj' should not be locked.", cooobj.objlock.objlocked);
@expect = "Test";
@actual = cooobj.objname;
%%ASSERT(@expected != @actual);
%%ASSERT("Expecting " + @expect + ", but actual value is '" + @actual +
  "'.", @expect, @actual);
```

12.10.7.4 %%DEBUGGER

The `%%DEBUGGER` directive can be used to set a breakpoint in a Fabasoft app.ducx Expression.

Syntax

```
%%DEBUGGER;
```

12.10.7.5 %%LOG

The `%%LOG` directive can be used to log messages to Fabasoft app.telemetry. App.telemetry provides the log level `IPC`, `NORMAL`, `DETAIL` and `DEBUG`.

Syntax

```
%%LOG(message);
%%LOG(message, level);
```

Example

```
%%LOG(cooobj.objlock.objlocked); //Detail level
%%LOG(cooobj.objlock.objlocked, "DEBUG"); //Debug level
```

12.11 Searching for objects – app.ducx Query Language

The app.ducx Query Language can be used to search for objects in Fabasoft Folio. The search always refers to an object class (and by default derived object classes). To carry out a search the Fabasoft Folio Runtime methods `SearchObjects` and `SearchObjectsAsync` can be used. `SearchObjects` returns the search result array at once (10,000 objects at the maximum) whereas `SearchObjectsAsync` returns a `searchresult`, which can be used to step through the result (without limit).

The following example shows a Fabasoft app.ducx expression that illustrates how to search for orders at once and asynchronously.

Example

```
integer @bulksize = 150;
string @query = "SELECT objname FROM APPDUCXSAMPLE@200.200:Order";


// Performs a search with SearchObjects
```

```
Order[] @results = coort.SearchObjects(cootx, @query);
%%TRACE("Number of hits", count(@results));


// Performs a asynchronous search with SearchObjectsAsync
searchresult @sr = coort.SearchObjectsAsync(cootx, @query);
Order[] @resultsAsync = null;
// Steps through the search result
while (count(@resultsAsync = @sr.GetObjects(@bulksize)) > 0) {
  %%TRACE("Fetched chunk of search results", @resultsAsync);
  integer @objcnt;
  for(@objcnt = 0; @objcnt < count(@resultsAsync); @objcnt++) {
    %%TRACE("Result entry", @resultsAsync[@objcnt].objaddress);
  }
}
```

A search query is built up by following parts:

- Options (optional)
  Options can be used to restrict the search. A description of available options can be found afterwards in this chapter.
- `SELECT` clause
  In the `SELECT` clause properties should be defined that are accessed later on because these properties of found objects are loaded in the cache. When accessing these objects no further server request is necessary to read the defined properties.
  `SELECT *` loads all properties in the cache and therefore should only be used if many properties are used further on.
- `FROM` clause
  Defines the object classes that should be searched for. Per default derived object classes are also included in the search result. If derived object classes should not be found use the property `objclass` in the `WHERE` clause. In the following example only folders are found and not for instance synchronized folders. **Example:** `SELECT objname FROM COODESK@1.1:Folder WHERE objclass = 'COODESK@1.1:Folder'`
- `WHERE` clause (optional)
  The `WHERE` clause is used to restrict the search result by defining conditions.

Syntax

```
{Options} SELECT Properties FROM Classes [WHERE Condition]
```

A complete reference of the grammar can be found in chapter 12.13 “Grammar of the app.ducx Query Language”.

12.11.1 Options

In most cases no options will be required.

- `LIMIT`
  Restricts the search result to the defined number of objects. This setting can only be used with `SearchObjects`. The maximum value is 10,000.
- `PRELOAD`
  In case of an asynchronous search the `PRELOAD` value defines how many objects are fetched in advance when stepping through the search result.
- `TIMEOUT`
  Restricts the search time to the specified value (seconds).
  Example: `TIMEOUT 3`

- `NOCHECK`
  By default it is checked whether the defined properties in the SELECT clause belong to the object classes in the FROM clause. This option disables the check.
- `NOEXEC`
  Only a syntax check of the search query takes place, but the search itself gets not executed.
- `NOHITPROPERTIES`
  In case of a full-text search several hit properties (hit rank, hit count, hit display) may be displayed in the search result. With this option no hit properties are returned.
  Note: A full-text search is triggered when using `CONTAINS` or `LIKE '%%something'` in the `WHERE` clause.
- `HITPROPERTIES`
  In case of a full-text search hit properties (`COOSYSTEM@1.1:contenthitrank`, `COOSYSTEM@1.1:contenthitcount`, `COOSYSTEM@1.1:contenthitdisplay`) can be displayed in the search result. This option can be used to define which hit properties are returned.
  Example: `HITPROPERTIES(COOSYSTEM@1.1:contenthitrank)`
- `IGNORECASE`
  A case-insensitive search is carried out, even if the search is configured as case-sensitive in the domain and database.
- Location
  If no location is specified the search is carried out in the COO stores of the user's local domain.
  - `LOCAL`
    Restricts the search to the COO stores of the user's local domain.
  - `GLOBAL`
    The search is carried out in all known domains.
  - `DOMAINS`
    Restricts the search to the defined domains (list of addresses of the domain objects).
    Example: `DOMAINS('COO.200.200.1.1','COO.200.200.1.7')`
  - `CACHE`
    Restricts the search to the kernel cache.
  - `TRANSACTION`
    Restricts the search to objects belonging to the current transaction.
  - `SCOPE`
    The scope allows to define a query scope object (reference or object address) that defines the location the search is carried out.
    Example: `SCOPE(#COOSYSTEM@1.1:LoginQuery),SCOPE('COO.1.1.1.2686')`
  - `SERVICES`
    Restricts the search to the defined COO services.
  - `STORES`
    Restricts the search to the defined COO stores.
  - `ARCHIVES`
    Restricts the search to the defined archive stores.

12.11.2 Properties

It is useful to define properties that are accessed later on because these properties of found objects are loaded in the cache. When accessing these objects no further server request is necessary to read the defined properties.
`SELECT *` loads all properties in the cache and therefore should only be used if many properties are used further on.

12.11.3 Classes

Objects of the defined object classes (and derived object classes) are searched. If derived object classes should not be found use the property `objclass` in the `WHERE` clause.

Example

```
SELECT objname FROM COOSYSTEM@1.1:User WHERE objclass = 'COOSYSTEM@1.1:User'
```

12.11.4 Condition

To restrict the search specify values for properties of the object class defined in the `FROM` clause. Following general rules apply:

- Fully qualified references are used to define the properties. `COOSYSTEM@1.1` may be omitted for properties belonging to this software component. It is good practice to start the reference with a period to make clear that the property belongs directly to the object and is not part of a compound type.
- Compound types can be accessed using a property path.
  Example: `.COOMAPI@1.1:emailinformation.COOMAPI@1.1:emailaddress`
- String constants are defined with double quotes " or single quotes '. Special characters like " and ' can be escaped with a backslash \.
- Dates have to be provided this way: `yyyy-mm-dd hh:mm:ss`

Example

```
WHERE userlogname LIKE 'A%' AND userroles.userrolepos = 'COO.200.200.1.7'
```

Following keywords can be used to specify a condition:

- `NOT`
  The expression yields the value `true` if the operand evaluates to `false`, and yields the value `false` if the operand evaluates to `true`.
- `AND`
  Indicates whether both operands are true.
- `OR`
  Indicates whether either operand is true.
- <, <=, >, >=, =, <>
  Compares two operands: less, less equal, greater, greater equal, equal, not equal
- `[SOUNDS] [NOT] LIKE`
  `LIKE` determines whether the left string matches the right string. The `%`, `*`, `?` and _ wildcards can be used in the right string operand. The `LIKE` operator can be preceded by the `SOUNDS` keyword for a phonetic comparison.
  Example: `WHERE COOMAPI@1.1:emailinformation.COOMAPI@1.1:emailaddress LIKE "*fabasoft.com"`
- `[NOT] CONTAINS`
  Triggers a full text search.
  Example: `WHERE COOSYSTEM@1.1:contcontent CONTAINS 'Workflow'`
- `[NOT] IN`
  Determines whether the value is in the defined list.
- `[NOT] INCLUDES`
  Determines whether the value of the right operand is an element of the list provided in the left operand.

- `[NOT] BETWEEN … AND …`
  Determines whether the value is between the specified boundaries.
- `IS [NOT] NULL`
  Determines whether the property has a value.
- `UPPER`
  Converts all characters of a property to upper case (string data type).
- `LOWER`
  Converts all characters of a property to lower case (string data type).
- `SUM`
  Calculates the sum of all property values (numeric data type).
- `AVG`
  Calculates the average of all property values (numeric data type).
- `COUNT`
  Calculates the number of elements of a property (any data type).
- `MIN`
  Calculates the smallest value of all property values (numeric, string, date data type).
- `MAX`
  Calculates the largest value of all property values (numeric, string, date data type).

12.11.5 Search query examples

The following example shows a variety of possibilities to define search queries.

Example

```
// Returns all Note objects
SELECT objname FROM NOTE@1.1:NoteObject
// Returns contact persons with "Jefferson" in COOSYSTEM@1.1:usersurname
SELECT objname FROM FSCFOLIO@1.1001:ContactPerson WHERE .usersurname = 'Jefferson'
// The settings in the query scope object restrict the search
// Account objects are returned that reference the current user as owner
SCOPE (#FSCFOLIOCRM@1.1001:CRMQueryScope) SELECT * FROM FSCFOLIOCRM@1.1001:Account WHERE
.objowner = coouser
// The search is restricted to the domain with object address COO.1.1900.1.1
DOMAINS ('COO.1.1900.1.1') SELECT .objname FROM COOSYSTEM@1.1:CurrentDomain
// Returns users that are created between the last hour and last half-hour
SELECT objname FROM COOSYSTEM@1.1:User WHERE
(.objcreatedat >= coonow-60*60)) AND
(.objcreatedat < coonow-30*60))
// Returns users with a task in the task list
SELECT objname FROM COOSYSTEM@1.1:User WHERE .COOAT@1.1001:attasklist IS NOT NULL
// A query scope object is used and the search is restricted to 100 result entries
coort.SearchObjects(cootx, "SCOPE (#FSCLEGALHOLD@1.1001:LegalHoldQueryScope) LIMIT 100
SELECT objname FROM Object WHERE .FSCLEGALHOLD@1.1001:objlegalholds = \"" +
cooobj.objaddress + "\"");
```

12.12 Grammar of the app.ducx Expression Language

The grammar of the app.ducx Expression Language is formally defined as described below.

Grammar

app.ducx Expression Language

```
Expression := { Statement }.
Statement := ( ";" | Directive ";" | Block | Declare ";" | If | For | While | Do ";" |
```

```
  Switch | Break ";" | Continue ";" | Return ";" | Yield ";" | Try | Throw ";" |
  Sequence [ ";" ] ).
Directive := "%%" Name [ ( Sequence | "(" [ Sequence ] ")" ) ].
Block := "{" Expression "}".
Declare := "DECLARE" [ ">" | "&" | "<" ] Ident { "," [ ">" | "&" | "<" ] Ident }.
If := "IF" "(" Cond ")" Block [ "ELSE" ( If | Block ) ].
For := "FOR" "(" Sequence ( ";" [ Cond ] ";" Sequence | ":" ( Sequence | Block ) ) ")"
  Block.
While := "WHILE" "(" Cond ")" Block.
Do := "DO" Block "WHILE" "(" Cond ")".
Switch := "SWITCH" "(" Cond ")" "{" { ( "CASE" ( Ident | Const ) | "DEFAULT" ) ":"
  [ Expression ] } "}".
Break := "BREAK".
Continue := "CONTINUE".
Return := "RETURN" [ Assign ].
Yield := "YIELD" Assign.
Try := "TRY" [ [ "NEW" ] "TRANSACTION" ] Block {
  "CATCH" "(" ( [ "+" | "-" ] Integer | Object |
    [ ( "@" | "::" | ":>" ) ] Ident | "..." ) ")" Block }
  [ "FINALLY" Block ].
Throw := "THROW" Cond.
Items := [ Init ] { "," [ Init ] }.
Sequence := [ Assign ] { "," [ Assign ] }.
Arguments := [ Assign ] { "," [ Assign ] }.
Init := Cond [ ( ":" | "=" ) Assign ].
Assign :=
  ( Cond ( [ [ "[]" ] [ ( "@" | "::" | ":>" ) ] Ident ]
  [ ( "=" | "&=" | "^=" | "|=" | "<<=" | ">>=" | "+=" | "-=" | "*=" | "/=" | "%=" )
    Assign ] |
  "->" Qualifier "(" Arguments ")" ) |
  "->" Qualifier [ "(" Arguments ")" ] ).
Cond := Or [ "?" Sequence ":" Cond ].
Or := And { ( "||" | "OR" ) And }.
And := BitOr { ( "&&" | "AND" ) BitOr }.
BitOr := BitXOr { "|" BitXOr }.
BitXOr := BitAnd { "^" BitAnd }.
BitAnd := Equal { "&" Equal }.
Equal := Rel [ ( "==" | "!=" | "<>" ) Rel ].
Rel := Shift [ ( ( "<" | "<=" | ">" | ">=" | "<=>" ) Shift |
  [ "SOUNDS" ] [ "NOT" ] "LIKE" Shift |
  [ "NOT" ] "CONTAINS" Shift |
  [ "NOT" ] "IN" Shift |
  [ "NOT" ] "INCLUDES" Shift |
  [ "NOT" ] "BETWEEN" Shift "AND" Shift |
  "IS" [ "NOT" ] "NULL" ) ].
Shift := Add { ( "<<" | ">>" ) Add }.
Add := Mul { ( "+" | "-" ) Mul }.
Mul := Prefix { ( "*" | "/" | "%" ) Prefix }.
Prefix := ( Postfix |
  ( "&" | "++" | "--" | "!" | "NOT" | "~" | "+" | "-" ) Prefix ).
Postfix := Primary { "." Qualifier |
  "(" Arguments ")" [ "[" ( "..." | Sequence ) "]" ] |
  "[" Sequence "]" |
  "++" | "--" }.
Qualifier := ( Ident | Reference | "[" Sequence "]" ).
Primary := (
  "@" ( "THIS" | Ident ) |
  "::" ( "THIS" | Ident | Reference ) |
  ":>" ( "THIS" | Ident | Reference ) |
  "THIS" | Ident | Reference |
  "[]" | "NULL" | "TRUE" | "FALSE" |
  "COORT" | "COOTX" | "COOOBJ" | "COOMETH" | "COOCONTEXT" |
  "COOUSER" | "COOENV" | "COOROOT" | "COOLANG" | "COODOMAIN" | "COONOW" |
  ( [ "UPPER" | "LOWER" | "COUNT" | "SUM" | "AVG" | "MIN" | "MAX" ] "(" Sequence ")" ) |
  "INSERT" "(" Assign "," Assign "," Assign ")" |
  "DELETE" "(" Assign "," Assign [ "," Assign ] ")" |
  "FIND" "(" Assign "," Assign ")" |
  "SORT" "(" Assign ")" |
  "UNIQUE" "(" Assign ")" |
  "REVERT" "(" Assign ")" |
  "SUPER" "(" ")" |
  "TYPEOF" "(" Assign ")" |
```

```
  "INDEXOF" "(" Assign "," Assign ")" |
  "STRLEN" "(" Assign ")" |
  "STRTRIM" "(" Assign ")" |
  "STRHEAD" "(" Assign "," Assign ")" |
  "STRTAIL" "(" Assign "," Assign ")" |
  "STRSPLIT" "(" Assign "," Assign ")" |
  "STRJOIN" "(" Assign [ "," Assign ] ")" |
  "STRREPLACE" "(" Assign "," Assign [ "," Assign ] ")" |
  "(" Sequence ")" |
  "{" Items "}" |
  "[" Sequence "]" |
  Const ).
Const := ( String | [ "+" | "-" ] Integer | [ "+" | "-" ] Float | DateTime | Object ).
Ident := ( "$" | [ "$" ] Name ).
Letter := "a" ... "z" "A" ... "Z".
Digit := "0" ... "9".
HexDigit := "0" ... "9" "a" ... "f" "A" ... "F".
Digits := Digit { Digit }.
HexDigits := HexDigit { HexDigit }.
Name := Letter { ( Letter | Digit | "_" ) }.
Reference := Name [ "@" Digits "." Digits [ ":" Name ] ].
String := ( "'" ... "'" | '"' ... '"' ).
Integer := ( Digits | "0" ( "x" | "X" ) HexDigits ).
Float := Digits "." [ Digits ]
  [ ( "d" | "D" | "e" | "E" ) [ ( "+" | "-" ) ] Digits ].
DateTime := Digits "-" Digits "-" Digits
  [ ( "T" | " " ) Digits [ ":" Digits [ ":" Digits ] ] ].
Object := ( Address | "#" Reference ).
Address := "COO." Digits "." Digits "." Digits "." Digits [ "@" DateTime ].
```

12.13 Grammar of the app.ducx Query Language

The grammar of the app.ducx Query Language is formally defined as described below.

Grammar

app.ducx Query Language

```
Statement := ( Query | Evaluation ).
Query := { Options } "SELECT" Properties
  ( "FROM" Classes | "EXECUTE" Procedure ) [ "WHERE" Condition ].
Evaluation := "EVALUATE" Sequence "WHERE" Condition.
Options := ( "LIMIT" Integer | "PRELOAD" Integer | "TIMEOUT" Integer |
  "NOCHECK" | "NORESTRICTIONS" | "NOEXEC" |
  "NOHITPROPERTIES" | "HITPROPERTIES" "(" Properties ")" |
  "IGNORECASE" | Location ).
Location := ( "CACHE" | "TRANSACTION" | "LOCAL" | "INSTALLATION" | "GLOBAL" |
  "SERVICES" "(" Service { "," Service } ")" |
  "SCOPE" "(" Scope ")" |
  "DOMAINS" "(" Domain { "," Domain } ")" |
  "STORES" "(" Store { "," Store } ")" |
  "ARCHIVES" "(" Archive { "," Archive } ")" ).
Properties :=
  ( "*" | Property { "," Property } ).
Classes :=
  Class { "," Class }.
Condition := Term { "OR" Term }.
Term := Factor { "AND" Factor }.
Factor := [ "NOT" ] Primary.
Primary := ( Predicate | "(" Condition ")" ).
Predicate := Expression
  [ ( ( "<" | "<=" | ">" | ">=" | "=" | "<>" ) Expression |
  [ "SOUNDS" ] [ "NOT" ] "LIKE" Shift |
  [ "NOT" ] "CONTAINS" Shift |
  [ "NOT" ] "IN" "(" ( Sequence | Query ) ")" |
  [ "NOT" ] "INCLUDES" "(" ( Sequence | Query ) ")" |
  [ "NOT" ] "BETWEEN" Shift "AND" Shift |
  "IS" [ "NOT" ] "NULL" ) ].
Expression := ( Identifier |
  [ ( "UPPER" | "LOWER" | "SUM" | "AVG" | "COUNT" | "MIN" | "MAX" ) ]
```

```
  "(" ( Identifier | Shift ) ")" |
  Shift ).
Property := ( Reference | Object ).
Class := ( Reference | Object ).
Procedure := ( Reference | Object ).
Identifier := [ "." ] { Reference "." } Reference ).
Domain := ( String | Object ).
Service := ( String | Object ).
Scope := ( String | Object ).
Archive := ( String | Object ).
Letter := "a" ... "z" "A" ... "Z".
Digit := "0" ... "9".
HexDigit := "0" ... "9" "a" ... "f" "A" ... "F".
Digits := Digit { Digit }.
HexDigits := HexDigit { HexDigit }.
Name := Letter { ( Letter | Digit | "_" ) }.
Reference := Name [ "@" Digits "." Digits [ ":" Name ] ].
String := ( "'" ... "'" | '"' ... '"' ).
Integer := ( Digits | "0" ( "x" | "X" ) HexDigits ).
Float := Digits "." [ Digits ]
  [ ( "d" | "D" | "e" | "E" ) [ ( "+" | "-" ) ] Digits ].
DateTime := Digits "-" Digits "-" Digits
  [ ( "T" | " " ) Digits [ ":" Digits [ ":" Digits ] ] ].
Object := ( Address | "#" Reference ).
Address := "COO." Digits "." Digits "." Digits "." Digits [ "@" DateTime ].
```

13 Testing and Debugging

This chapter covers testing and debugging of Fabasoft app.ducx projects, and introduces you to the Fabasoft software products that support you in the testing and debugging phase of the software development life cycle.

Fabasoft app.ducx includes built-in tracing capabilities for diagnosing problems, and lets you use the debugging facilities of your development environment to debug your source code in a convenient and efficient manner.

The Fabasoft app.ducx unit test allows you to automate the testing process when doing unit testing to ensure that your code still works correctly as you change it.

The integration of Fabasoft app.test allows you to automate the testing process when doing user interface testing to ensure that your code changes applies to the user interface correctly as you change it.

Fabasoft app.telemetry provides a highly advanced measuring and profiling software which allows you to record the activities of your software application on each component of the Fabasoft reference architecture.

13.1 Tracing in Fabasoft app.ducx projects

The built-in tracing functionality of Fabasoft app.ducx allows you to produce extensive traces of your use case implementations.

To enable tracing, you can select several *Trace* modes in the project preferences dialog box of Eclipse. Trace messages are written to the Fabasoft app.ducx Tracer, which can be found in the `Setup\ComponentsBase\Trace` folder on your product DVD.

With trace mode “Trace Errors” enabled, errors that occur during runtime are traced.

With trace mode “Trace Expressions” enabled, `%%TRACE`, `%%FAIL` and `%%ASSERT` directives are evaluated for your software component.

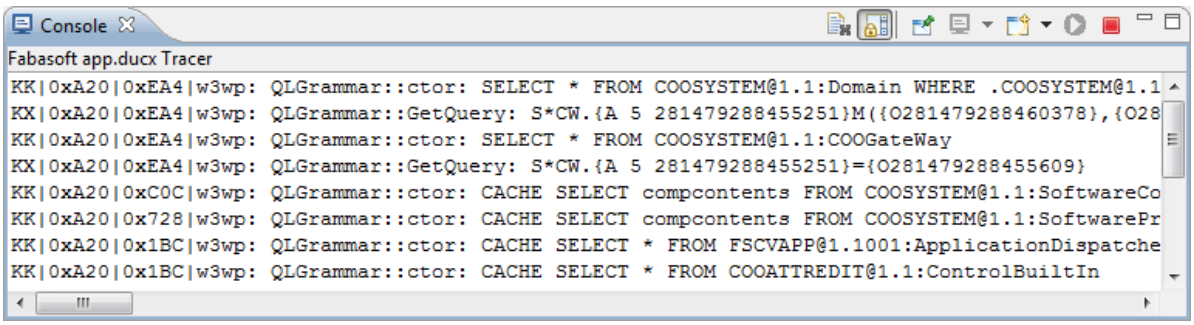
With trace mode “Trace Calls” enabled, the invocation of each use case of your app.ducx project produces extensive trace output.

Note: For Fabasoft Folio Cloud Apps, all available trace modes can be enabled with “enable tracing” in the projects preferences dialog box of Eclipse.

The trace output contains detailed information about all use cases invoked by your use case implementations, along with the parameter values passed to and returned by the invoked use cases to allow you to get a complete picture of the call stack.

13.2 Display Trace messages in Eclipse

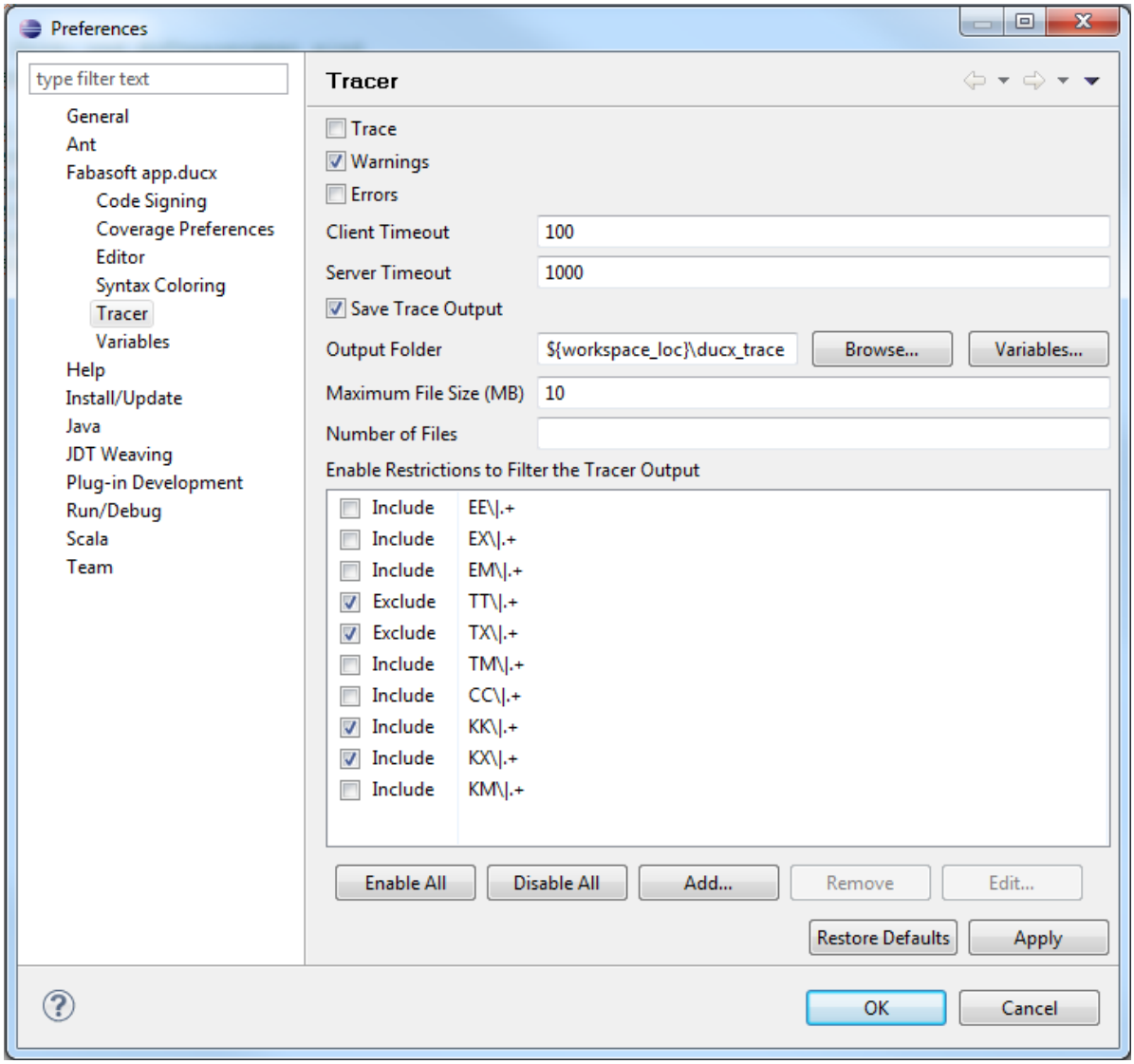
Fabasoft app.ducx extends the Console View to display trace messages from the default web service.



In the Eclipse preferences the trace output level can be customized. The client timeout defines the timespan the client waits until it sends the next request to the server. The server timeout defines the timespan the server waits maximally until the response is sent. Lower values increase the network traffic.

Saving the trace messages can be enabled. The output folder, the maximum size of each output file and the number of files can be specified. When an output file has the given maximum size a new one is created, when the given number of files is reached the oldest one is deleted. The newest file is always `trace0.log`. A blank “Number of Files” field means infinite number of files.

Tracer messages can be filtered by Java Regular Expressions. You can define a set of regular expressions and whether the trace messages matching the expressions should be included or excluded. When no filter restriction is enabled, every message is included and nothing is excluded. The specified restrictions can be enabled or disabled.



13.3 Coverage

Code Coverage is a measure known from software testing. It describes the degree to which source code of a component has been tested, e.g. by executing a unit tests or an app.tests.

When compiling a Fabasoft app.ducx project, any relevant line is recorded and the resulting metadata will be a part of your component. Coverage can be measured, when tests are executing within a coverage session.

A coverage session can be started interactively or using ant tasks. Currently pure line coverage is used.

A coverage session will live as long until it is explicitly stopped. Sessions not stopped at service shutdown will be persisted and restarted at service startup. Coverage data files are provided for every software component.

The ratio between all touched lines and executed lines is your coverage.

13.3.1 Expression Coverage

Any Fabasoft app.ducx expression contained in a Fabasoft app.ducx project is instrumented by recording all executable lines. When executing within a coverage session, all executed Fabasoft app.ducx expression lines are touched.

13.3.2 Model Coverage

Currenlty Object class properties are instrumented. Model Coverage is based on accessing object properties. Every time a property of an object class is accessed, the corresponding source code line is touched.

Note: Fabasoft app.ducx Model Coverage is available from Summer Release 2012.

13.4 Debugging Fabasoft app.ducx projects

Fabasoft app.ducx allows you to debug use case implementations written in Java or Fabasoft app.ducx Expressions from within your development environment.

13.4.1 Debugging the Java implementation of an Fabasoft app.ducx project

Before you can debug an app.ducx project in Eclipse, make sure that the `COOJAVA_JVMOPTIONS` environment variable is set to `-agentlib:jdwp=server=y,transport=dt_socket,address=8000,suspend=n`. You may also use a port other than "8000" by setting the `address` parameter to the desired value. Keep in mind, that only one JVM can be bound to the same socket, otherwise the JVM will not be initialized.

Note: You can also use the `COOJAVA_JVMOPTIONS` environment variable to set other parameters for the Java virtual machine, e.g. the `-Xms` and `-Xmx` parameters for defining the size of the heap space. Adjusting the size of the heap space might be necessary if you run into the following Java runtime error:

```
java.lang.OutOfMemoryError: Java heap space
```

The `COOJAVA_JVMOPTIONS` environment variable can also be individually defined for each web service instance in order to avoid conflicts if multiple web services are hosted on a single machine.

To define the `COOJAVA_JVMOPTIONS` environment variable for a particular web service instance, add the following entries to the registry:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Fabasoft\FscWeb\Modules\FSC]
@="C:/Program Files/Fabasoft/Components/Web/FSC/ASP/content/bin/fscvext.dll"

[HKEY_LOCAL_MACHINE\SOFTWARE\Fabasoft\FscWeb\Modules\FSC\COOJAVA_JVMOPTIONS]
@="-agentlib:jdwp=server=y,transport=dt_socket,address=8000,suspend=n"
```

Note: If you are manually adding the described entries to the registry using the Registry Editor, you have to create the keys that do not exist yet. The "@" character in the example refers to the "(Default)" registry value.

Adapt the path to the virtual directory associated with your web service as appropriate. The correct path can be obtained by carrying out the following steps:

1. Start the Fabasoft app.ducx Tracer.
2. Restart the Fabasoft Folio Web Service.
3. Point your web browser to the Fabasoft Folio Web Service.
4. Search the trace output for the modulepath variable. Copy the value into the default registry value of the registry key `[HKEY_LOCAL_MACHINE\SOFTWARE\Fabasoft\FscWeb\Modules\FSC]`.

After setting the environment variable, the Fabasoft Folio Web Service must be restarted. If you are using Microsoft Internet Information Services to host the Fabasoft Folio Web Service, do not issue an `iisreset` command but restart the “World Wide Web Publishing Service” in the “Services” snap-in instead.

In order to verify that the remote debugging listening port has been installed correctly issue `netstat -a | more` and search the output for the specified listening port (e.g. 8000).

Furthermore, you have to create a new debug launch configuration in Eclipse. To do this, select the *Debug* from the *Run* menu. This will bring up the dialog box depicted in the next figure. Select “Remote Java Application” and click *New Launch Configuration*. Enter a *Name* for the debug launch configuration, select the *Project*, enter the name of the web server hosting the Fabasoft Folio Web Service in the *Host* field, click *Apply* to save your settings, and *Debug* to attach the remote debugger to the remote Java virtual machine.

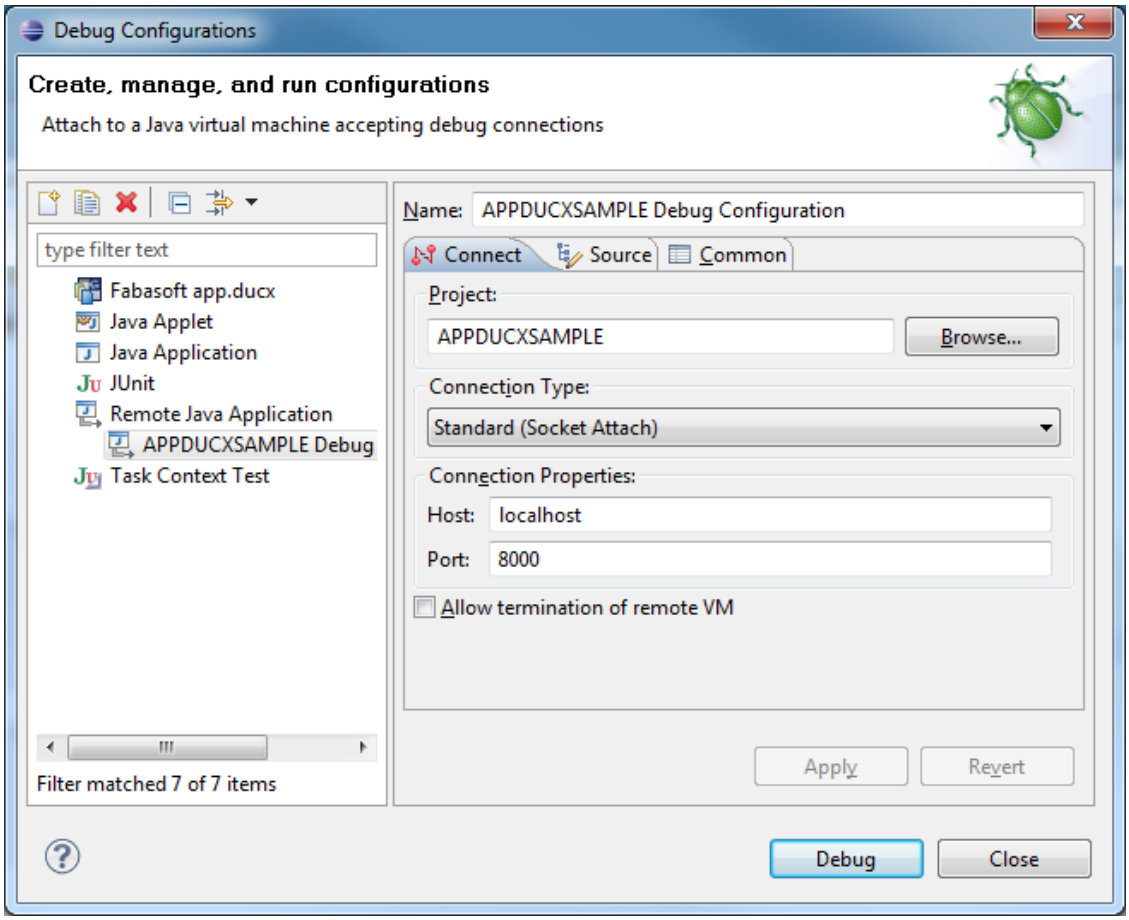


Figure 25: Creating a new debug launch configuration in Eclipse

Once a debug launch configuration has been created, you can start debugging your Java use case implementations.

You can set breakpoints in your Java code by clicking *Toggle Breakpoint* in the context menu in the corresponding line of code. When you click *Debug* from the *Run* menu, select your debug launch configuration and click *Debug* to run your Fabasoft app.ducx project. Execution will stop when a breakpoint is reached to take you to the Eclipse Debugger (see next figure).

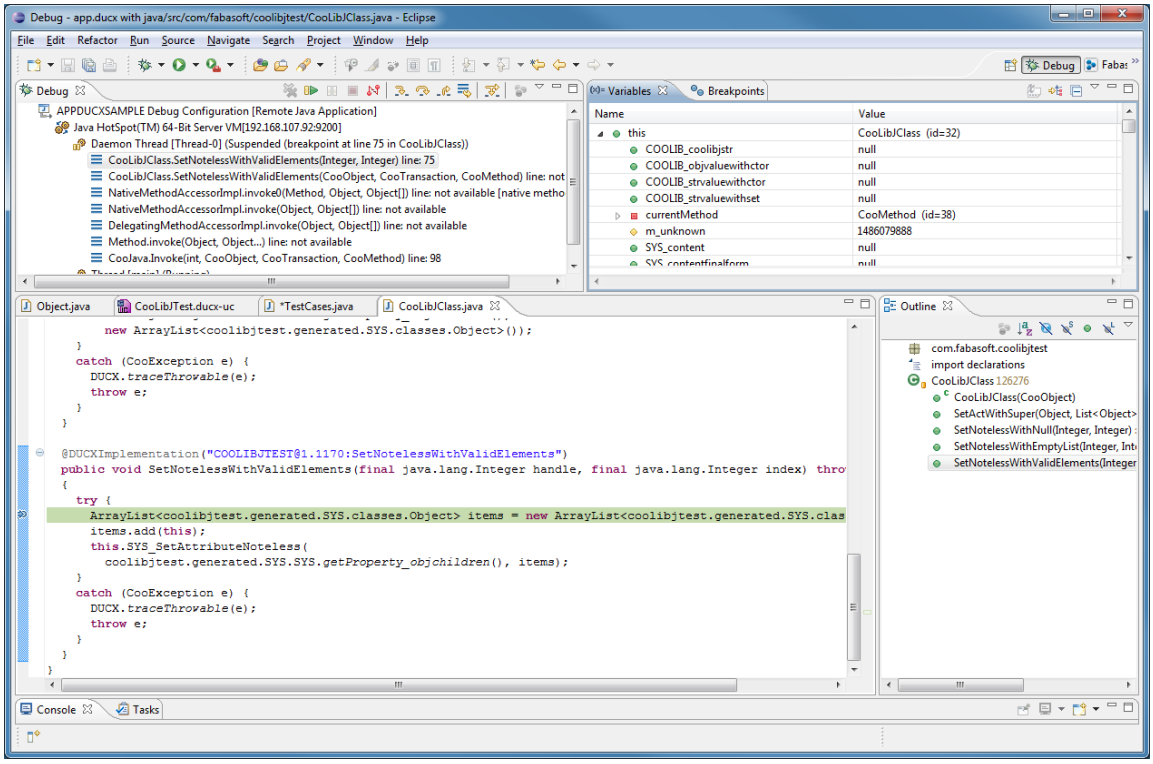


Figure 26: Debugging a Java use case implementation in Eclipse

For further information on how to debug Java projects in Eclipse, please refer to [AnLe10].

13.4.2 Debugging the Fabasoft app.ducx Expression implementation of an app.ducx project

13.4.2.1 Debug configurations

You have to create a new debug configuration before you can debug an Fabasoft app.ducx project in Eclipse. To do so, click “Debug Configurations“ on the “Run” menu.

In the dialog box, select “Fabasoft app.ducx” click the “New launch configuration” symbol and enter a *Name* for the new launch configuration and select the *Project* by clicking “Choose“. To specify how the Fabasoft app.ducx project should be debugged, click the “Debug” tab. Following options are available:

- *Attach*
  If this check box is selected, the Fabasoft app.ducx project is not uploaded before the debugging session starts.
- *Suspend*
  - *On error*
    If this check box is selected, the execution stops on errors.
  - *On warning*
    If this check box is selected, the execution stops on warnings.
  - *On commit*
    If this check box is selected, the execution stops on commits of transactions.
  - *On abort*
    If this check box is selected, the execution stops on aborts of transactions.

- *On query*
  If this check box is selected, the execution stops when a search query is executed.
- *Block until command*
  If this check box is selected, the execution stops on break points.

- *Transport*
  - *Include result nodes*
    If this check box is selected, the debug output is rich in detail. Keep in mind that the data to be transferred may be extensive.

Click “Apply” to save your settings, and “Debug” to debug the Fabasoft app.ducx project.

13.4.2.2 Breakpoints

Breakpoints can be set in every app.ducx Expression block with one of the following methods:

- Insert the `%%DEBUGGER;` directive in the Fabasoft app.ducx Expression.
- Double-click on the left margin of the source code editor.
- Select the desired line and use the shortcut `Ctrl + Shift + B`.
- Open the “Run” menu and click “Toggle Breakpoint”.

Note: Microsoft Windows: To extend the time a thread is kept in the application pool when waiting on a break point, set the `(Default)` value to `2147483647` of following registry key:

`[HKEY_LOCAL_MACHINE\SOFTWARE\Fabasoft\FscWeb\Modules\FSC\FSCVEXT_MAXREQUESTMSECS]`
In normal circumstances this should not be necessary.

13.4.2.3 Debug perspective

The debug information is displayed in the debug perspective. To switch to the debug perspective, open the “Window” menu, point to “Open Perspective” and click “Other”. Select “Debug” and click “OK”. The Debug perspective provides you debugging information like the call stack, values of variables and so on.

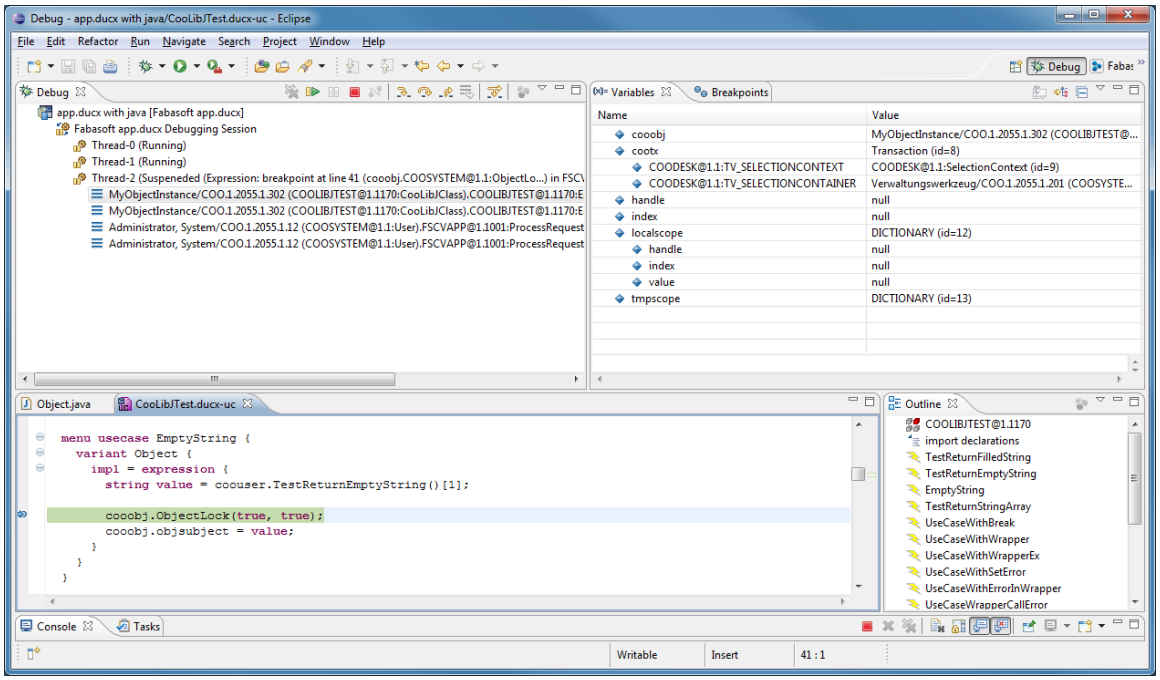


Figure 27: Debugging an app.ducx Expression use case implementation in Eclipse

Following common debug commands are available:

- “Resume”
  Resumes the execution until the next thread suspending event like a break point is reached.
- “Terminate”
  Terminates the debugging session and the suspended threads resume.
- “Step Into”
  The currently selected line is executed and the thread suspends on the next executable line of the called method, if the method is also implemented as Fabasoft app.ducx Expression within the same Fabasoft app.ducx project. Otherwise “Step Over” is carried out.
- “Step Over”
  The currently selected line is executed and the thread suspends on the next executable line of the Fabasoft app.ducx Expression.
- “Step Return”
  The currently selected line is executed and the thread suspends on the next executable line of the calling Fabasoft app.ducx Expression.

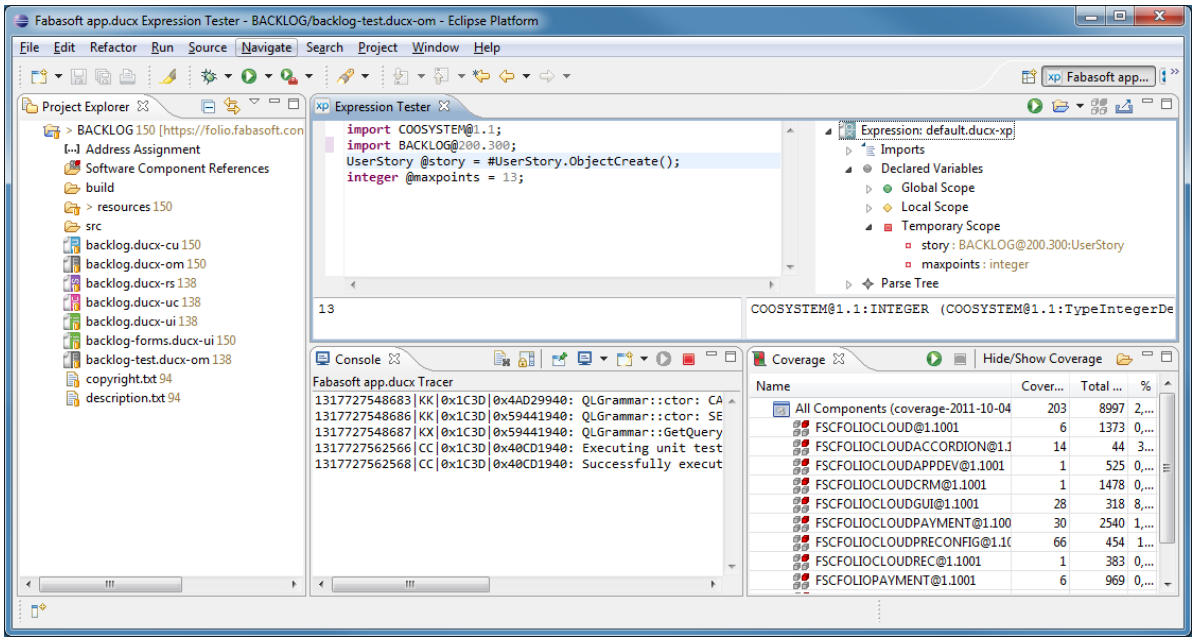
13.5 Expression Tester

The Expression Tester allows validating an expression against the default web service configured in the Eclipse preferences.

To use short references imports are to be done as in all other domain specific files. To do this, software references have to be selected and downloaded with the “Add/update references” button into the Expression Tester cache location or simply use a project context with the “Select context” button.

An expression can be validated with the “Evaluate expression at default web service” button or the shortcut `ALT + X`. Then a result is displayed in the bottom left box and the result’s type at the bottom right box.

On the right side, the imports and the declared variables are displayed.



13.6 Testing use cases with unit tests

The Fabasoft app.ducx unit test allows you to automate the testing process using unit tests to ensure that your code still works correctly as you change it.

Syntax

```
instance UnitTest reference {
  test = expression {...}
  testdata<tdid,tdcontent> = {...}
}
instance UnitTestGroup reference {
  tests = {...}
}
```

A unit test is an instance of the object class `FSCDUCXUNIT@1.1001:UnitTest` and identified by a reference. A unit test holds in the property `test` an expression which contains the unit test. In this expression, private component objects of other software components can be used. The aggregate `testdata` consists of the string property `tdid` and the content property `tdcontent`.

A unit test group is an instance of the object class `FSCDUCXUNIT@1.1001:UnitTestGroup` and identified by a reference. A unit test group holds in the property `tests` a list of unit tests. The order of the test execution depends on the order of definition in the property `tests`.

Example

```
objmodel APPDUCXSAMPLE@200.200
{
  import COOSYSTEM@1.1;
  import FSCDUCXUNIT@1.1001;
  instance UnitTest TestContentExtraction {
    test = expression {
      content zipfile = #TestContentExtraction.testdata[tdid == "zip”].tdcontent
      string extractdir = cooobj.ExtractZipFile(zipfile);
      %%ASSERT(strlen(extractdir) > 0);
    }
    testdata<tdid,tdcontent> = {
      { "zip", file("resources/data.zip") }
      ...
      }
  }
  instance UnitTestGroup TestGroupExtractions {
    tests = {
      TestContentExtraction,
      TestContentExtractionExtended
    }
  }
}
```

In a Fabasoft Folio Domain unit tests can be started manually. To do so, click *Run Test* on the context menu of the unit test or *Run Tests* on the context menu of the unit test group object.

Unit tests can be integrated into an automated build and test environment using Fabasoft app.ducx Ant tasks. Information about Ant tasks can be found in chapter 3.5.4 “Fabasoft app.ducx Ant tasks”.

13.7 Testing use cases with Fabasoft app.test

Fabasoft app.test allows testing implemented use cases in a convenient way. Recording and playing of interactions happens within a context-sensitive user interface. This means that rather than simply recording mouse moves, clicks and keyboard strokes, objects in the user interface are called instead.

For further information on Fabasoft app.test and to learn how to define app.test use cases, please refer to [Faba10b].

13.8 Fabasoft app.telemetry

Fabasoft app.telemetry is a revolutionary measuring and profiling software. This tool allows you to record the activities of any software application at runtime spanning all layers of the Fabasoft Folio Reference Architecture.

Using the collected data, you can easily analyze the processing steps of requests to a Fabasoft Folio Web Service throughout all Fabasoft Folio Services involved.

For the collection of data as requests are processed, so-called measuring points have been added to all Fabasoft Folio Services and the Fabasoft Folio Kernel. These measuring points remain deactivated as long as the recording of request data is not initiated using the Fabasoft app.telemetry Control Center Management Interface.

The seamless integration of Fabasoft app.test allows for a detailed analysis of app.test use cases and use case sequences using the Fabasoft app.telemetry Analyzer tool.

Fabasoft app.telemetry is an essential tool for every software component developer as it allows you to evaluate the effects of hardware, software and configuration changes.

The Fabasoft app.telemetry Analyzer combines the data of different services to give you an overview of the performance impact of processed requests. You can then drill down throughout all levels of processing. This way, you can track performance issues down to the use case level so you can optimize use case implementations. Moreover, the Fabasoft app.telemetry Analyzer allows you to drill down even further so you can even see the SQL statements issued on the database server.

14 Appendix

14.1 Comprehensive Java example

The following example demonstrates the implementation of a use case for importing customer orders from an XML document. The XML document is stored in a structure of compound type `COOSYSTEM@1.1:Content` which is passed to the use case as an input parameter. An order object is created in Fabasoft Folio for each of the order records found in the XML document. Afterwards, the orders' properties are populated with data from the XML records. Finally, the new order objects are added to the customer's list of orders.

Example

XML File

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<orders>
  <order>
    <orderid>1000251</orderid>
    <orderdate>22.01.2010</orderdate>
    <orderpositions>
      <orderposition>
        <productid>J82-K32-629</productid>
        <quantity>10</quantity>
      </orderposition>
      <orderposition>
        <productid>A72-G53-882</productid>
        <quantity>2</quantity>
      </orderposition>
    </orderpositions>
  </order>
  <order>
```

```
    <orderid>1000252</orderid>
    <orderdate>23.01.2010</orderdate>
    <orderpositions>
      <orderposition>
        <productid>B03-X53-341</productid>
        <quantity>1</quantity>
      </orderposition>
    </orderpositions>
  </order>
</orders>
```

app.ducx Use Case Language

```
usecase ImportOrders(Content xmlcontent) {
  variant Person {
    impl = java:APPDUCXSAMPLE.Person.ImportOrders;
  }
}
```

Java Source Code

```
package APPDUCXSAMPLE;
import java.io.File;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import static APPDUCXSAMPLE.generated.DUCX.coort;
import APPDUCXSAMPLE.generated.DUCX;
import APPDUCXSAMPLE.generated.DUCXImplementation;
import APPDUCXSAMPLE.generated.COOSYSTEM_1_1.structs.Content;
import APPDUCXSAMPLE.generated.APPDUCXSAMPLE_200_300.APPDUCXSAMPLE_200_300;
import APPDUCXSAMPLE.generated.APPDUCXSAMPLE_200_300.classes.Order;
import APPDUCXSAMPLE.generated.APPDUCXSAMPLE_200_300.classes.Product;
import APPDUCXSAMPLE.generated.APPDUCXSAMPLE_200_300.structs.OrderPosition;
import CooLib.CooObject;
import CooLib.CooContent;
public class Person extends APPDUCXSAMPLE.generated.FSCFOLIO_1_1001.
  classes.Person {
    public Person(CooObject obj) {
      super(obj);
  }
  private String getXMLValue(Node parentnode, String tagname) {
    Element elmnt = (Element) parentnode;
    NodeList nodelst = elmnt.getElementsByTagName(tagname);
    Element childelmnt = (Element) nodelst.item(0);
    return childelmnt.getChildNodes().item(0).getNodeValue();
  }
  @DUCXImplementation("APPDUCXSAMPLE@200.200:ImportOrders")
  public void ImportOrders(final Content xmlcontent) throws Exception {
    try {
      // Build XML DOM from XML content
      CooContent content = xmlcontent.COOSYSTEM_1_1_contcontent;
      File file = new File(content.GetFile("", false));
      DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
      DocumentBuilder db = dbf.newDocumentBuilder();
      Document doc = db.parse(file);
      doc.getDocumentElement().normalize();
      NodeList ordernodelst = doc.getElementsByTagName("order");
      for (int i = 0; i < ordernodelst.getLength(); i++) {
        // Initialize the list of order positions
        ArrayList<OrderPosition> poslst = new ArrayList<OrderPosition>();
        Node ordernode = ordernodelst.item(i);
        if (ordernode.getNodeType() == Node.ELEMENT_NODE) {
```

```
          // Retrieve order date from XML
          String orderDatestr = getXMLValue(ordernode, "orderdate");
          SimpleDateFormat dateformat = new SimpleDateFormat("dd.MM.yyyy");
          Date orderdate = dateformat.parse(orderDatestr);
          // Retrieve list of order positions from XML
          Element orderelmnt = (Element) ordernode;
          Element orderpositionselmnt = (Element) orderelmnt.
            getElementsByTagName("orderpositions").item(0);
          NodeList orderpositionnodelst = orderpositionselmnt.
            getElementsByTagName("orderposition");
          // Iterate through order positions (an order may consist of one
          // or more order positions)
          for (int j = 0; j < orderpositionnodelst.getLength(); j++) {
            Node orderposnode = orderpositionnodelst.item(j);
            if (orderposnode.getNodeType() == Node.ELEMENT_NODE) {
              // Retrieve product ID for current order position
              String productid = getXMLValue(orderposnode, "productid");
              // Retrieve quantity
              Long orderquantity = new Long(getXMLValue(orderposnode,
                "quantity"));
              // Find existing product object matching the product ID
              // in Fabasoft Folio
              CooObject[] orderproducts = coort.SearchObjects(DUCX.
                cootx, "LIMIT 1 SELECT objname FROM APPDUCXSAMPLE@200.200:
                Product WHERE .APPDUCXSAMPLE@200.200:productid = \"" +
                productid + "\"");
              if (orderproducts.length > 0) {
                Product orderproduct = Product.from(orderproducts[0]);
                if (orderproduct.isValid() && orderquantity != null) {
                  // Create new order position item and add it to the
                  // list of order positions
                  OrderPosition item = OrderPosition.create();
                  item.APPDUCXSAMPLE 200 300 product = orderproduct;
                  item.APPDUCXSAMPLE_200_300_quantity = orderquantity;
                  poslst.add(item);
                }
                else {
                  coort.SetError(APPDUCXSAMPLE_200_300.
                  getErrorMessage InvalidOrder());
                }
              }
              else {
                coort.SetError(APPDUCXSAMPLE_200_300.
                getErrorMessage ProductNotFound());
              }
            }
          }
          // Create new order object in Fabasoft Folio
          Order order = Order.create();
          // Set order date and positions to order object
          order.APPDUCXSAMPLE_200_300_orderdate = orderdate;
          order.APPDUCXSAMPLE_200_300_orderpositions = poslst;
          // Add the new order to the customer's list of orders
          order.COODESK_1_1_ShareObject(null, null, APPDUCXSAMPLE_200_300.
            getProperty_customerorders(), this);
        }
      }
    } catch (Exception e) {
      coort.Trace("Exception occurred: " + e.getMessage());
      throw e;
    }
  }
}
```

14.2 Creating a wizard

The following example shows how to implement a simple wizard for creating a group.

Example

app.ducx Object Model Language

```
// The graddmemebers property is used to add members to the group
// within the second wizard step
extend class Group {
  unique Object[] graddmembers {
    allow {
      User create;
    }
  }
}
```

app.ducx User Interface Language

```
// The form contains the two form pages that are displayed in the first
// and second wizard step
form FormGroupCreateGroupWizApp {
  audience = enduser;
  formpage PageGroupCreateGroupWizApp {
    audience = enduser;
    dataset {
      grlongname;
      grshortname;
      grsupergroups;
      grorgunittype;
      objexternalkey;
    }
  }
  formpage PageMembersCreateGroupWizApp {
    audience = enduser;
    dataset {
      graddmembers;
    }
  }
}
```

app.ducx Use Case Language

```
override InitializeCreatedObject {
  variant Group {
    impl = application {
      expression {
        Object venv_object;
        Object venv_parent;
        integer venv_index;
        Object venv_view;
        Object venv action;
        WizardContext[] @venv wizardctx;
        Object @group;
        venv_object.ObjectLock(true, true);
        if (venv_parent.HasClass(#Group) && venv_view == #grsubgroups) {
          venv_object.grsupergroups = venv_parent;
        }
        @group = venv object;
        // Creates the wizard context
        @venv_wizardctx = [
         {@group, #FormGroupCreateGroupWizApp, #ShowWizardApp}
        ];
        // Calls the wizard; the parameters are defined in the
        // FSCVENV@1.1001:WizardPrototype
        ->DoWizardApp(venv_object, venv_parent, venv_index, venv_view,
            venv_action, @venv_wizardctx);
        venv object.AddUserRole(venv object.graddmembers, #StaffPos,
            venv object);
        venv_object.graddmembers = null;
      }
    }
  }
}
// When creating a group the implementation of InitializeCreatedObject
```

```
// gets executed
override InitializeCreatedObjectDoDefault {
  variant Group {
    // GroupInitializeCreatedObject is implicitely generated
    // from InitializeCreatedObject with variant Group
    impl = GroupInitializeCreatedObject;
  }
}
```

The wizard context is a compound property list, used to define the sequence of the wizard. The compound type `FSCVENV@1.1001:WizardContext` consists of following properties:

- `FSCVENV@1.1001:wizardobject`
  Object containing the data.
- `FSCVENV@1.1001:wizardform`
  Form or form page filtered according user profile and access check (`AccTypeSearch`).
- `FSCVENV@1.1001:wizardapplication`
  Application used for this wizard step. You can use the generic implementation `FSCVENV@1.1001:ShowWizardApp`, or create your own application with the proper prototype. Inside the application it is up to you to call `FSCVENV@1.1001:ShowWizardApp` to display the form pages.
- `FSCVENV@1.1001:wizardpageidx`
  This parameter is used internally. It stores the form page visited lastly.
- `FSCVENV@1.1001:wizardpages`
  This parameter is used internally. It stores the form pages that should be displayed depending on the access check and user profile.
- `FSCVENV@1.1001:wizardcxtvisited`
  This parameter is used internally. It stores whether this context line has been visited before.
- `FSCVENV@1.1001:wizardreadonly`
  This parameter defines whether the form pages defined in this context line should be displayed read-only.
- `FSCVENV@1.1001:wizardoptional`
  This parameter defines whether this context line and all following context lines are optional. If a context line is optional a finish branch will be displayed.

14.3 Not translated object classes

Object class	Domain specific construct
COOAR@1.1:ActiveReport_Text	ActiveReport_Text
COOAR@1.1:ActiveReport_Web	ActiveReport_Web
COOAR@1.1:ActiveReportingAction	ActiveReportingAction
COOAR@1.1:ContentEvaluationAction	ContentEvaluationAction
COOATTREDIT@1.1:ButtonAction	ButtonAction
COOATTREDIT@1.1:DeskForm	**deskform**
COOATTREDIT@1.1:DisplayItem	DisplayItem

COOATTREDIT@1.1:FormPage	`formpage`
COOATTREDIT@1.1:ObjectEditForm	ObjectEditForm
COODESK@1.1:MenuRoot	`menuroot`
FSCCALDAV@1.1001:VTimezone	VTimezone
FSCOWS@1.1001:SOAPAction	SOAPAction
FSCVAPP@1.1001:Application	Application or automatic generated reference thru a `menu usecase`
COOSYSTEM@1.1:Action	<Action reference>
COOSYSTEM@1.1:MethodDefinition	MethodDefinition
COOSYSTEM@1.1:Prototype	Prototype
COOSYSTEM@1.1:UseCase	`usecase`
COOSYSTEM@1.1:TypeAggregateDef	`struct`
COOSYSTEM@1.1:TypeEnumDef	`enum`
COOSYSTEM@1.1:TypeCustomizationPointDef	<Customization definition reference>
COOXML@1.1:XMLSchema	XMLSchema
FSCOWS@1.1001:WebServiceDefinition	WebServiceDefinition

14.4 Transformation of legacy software components

14.4.1 Object Model Language

Language Construct	Transformer support
Class/Class extension	Supported
Struct/Struct extension	Supported
Enum/Enum extension	Supported
Instance/Instance extension	Supported
Relation	Generated as class and properties
Field	Supported
Property initialization values	Supported Supported

constraints	Supported
backlink	Supported

14.4.2 Resource Language

Language Construct	Transformer support
Error message	Supported
String	Supported
Symbol	Supported

14.4.3 User Interface Language

Language Construct	Transformer support
Form/Deskform dataset layout	Supported Supported Ignored
Form/Deskform extension	Supported
Formpage/Formpage extension	Supported
Menu/Menu extension	Generic Generated as an instance or a set of instances
Menu root	Supported
Portal	Generic
Taskpane/Taskpane extension	Supported
Binding	Supported
Button	Generic
Buttonbar/Buttonber extension	Supported/ Commented, when in backend is not available anymore

14.4.4 Use Case Language

Language Construct	Transformer support
Transaction Variable	Generated as enum and properties for `COOSYSTEM@1.1@TransactionVariables`

Usecase/Action	Supported
Override	Supported
Menu Usecase	Supported
Wrappers	Generic
Virtual Application	Generic

14.4.5 Organization Structure Language

Language Construct	Transformer support
Position	Supported
Organizational Unit/Organizational Unit extension	Supported
Access type	Supported

14.4.6 Business Process language

Language Construct	Transformer support
Activity/Activity extension	Supported
Process	Partially supported: conditions/loops/case construct might result errors

14.4.7 Customization Language

Language Construct	Transformer support
Customization	Generic
Customization Point	Generic

14.4.8 Expression Language

In expressions short references are used whenever possible. If parse errors occur, no changes are performed and the expression will be taken as it is.

14.5 Base App Profile Restrictions

14.5.1 Object Model Language

Restriction	Restricted element	Severity

Only Compound-, Basic-, ContentObject own and friend classes can be used as base classes!	Class	Error
Extending external element from non-friend component is not allowed! Exception: `COOSYSTEM@1.1:UserEnvironment`	Class/Enumeration/Compound Type extension	Error
Extension with address must not be used!	Class/Enumeration/Compound Type extension	Error
Edition and/or Solution instance must not be used!	Instance	Error
Application must not be instantiated!	Instance	Error
Instances with address must not be used!	Instance	Error
Not component instance must not be used!	Instance	Error

Extending external instance from non-friend component is not allowed!
Exceptions:
`FSCFOLIOCLOUD@1.1001:trchildren`
`FSCFOLIO@1.1001:EC_Root`
Configuration instances
Instances of classes with `COOTC@1.1001:AppCategory` base class
	Instance extension	Error

Configuration instances must not reference external class from non-friend component!	Configuration instance extension	Error
AppCategory instances may only assign the apps and templates properties with apps and classes from own or friend components!	AppCategory instance extension	Error
In the class chain of <instance> <class> could not be resolved. Add component <components> to project references! If this restriction fails, the other restriction evaluations on instances and instance extensions might return incorrect results.	Instance, Instance extension	Error

14.5.2 User Interface Language

Restriction	Restricted element	Severity

External ui element from non-friend component should not be extended!	Form/Property Page/Desk Form/ Menu/Menu Root/Task Pane extension Binding	Warning
Extension with address must not be used!	Form/Property Page/Desk Form/ Menu/Menu Root/Task Pane extension Binding	Error

14.5.3 Use Case Language

Restriction	Restricted element	Severity
Overriding external action from non-friend component of external classes from non-friend component is not allowed!	Overrides	Error
Wrapper must not be used!	Use Case Wrapper	Error
Extending external element from non-friend component is not allowed!	Dialog Extension	Error
Extension with address must not be used!	Dialog Extension	Error

14.5.4 Organization Structure Language

Restriction	Resticted element	Severity
Defining organizational structure model is not allowed!	Organizational Structure Model	Error

14.5.5 Business Process language

Restriction	Resticted element	Severity

Defining positions and/or organizational units for activities is not allowed!	Activity	Error
Extending external element from non-friend component is not allowed!	Activity Extension	Error
Extension with address must not be used!	Activity Extension	Error
The use of condition, case statement, loop and wait element is not allowed!	Process	Error

14.5.6 Customization Language

Restriction	Resticted element	Severity
Adding customizations to a target other than EditionFolioCloud@1.1 is not allowed!	Target Definition	Error
Overriding target is not allowed!	Target Definition	Error

14.6 Cloud App Profile Restrictions

14.6.1 Object Model Language

Restriction	Restricted element	Severity

Only Compound-, Basic-, ContentObject own and friend classes can be used as base classes!	Class	Error
Extending external element from non-friend component is not allowed!	Class/Enumeration/Compound Type extension	Error
Extension with address must not be used!	Class/Enumeration/Compound Type extension	Error
Edition and/or Solution instance must not be used!	Instance	Error
Application must not be instantiated!	Instance	Error
Instances with address must not be used!	Instance	Error
Not component instance must not be used!	Instance	Error

Extending external instance from non-friend component is not allowed!
Exceptions:
`FSCFOLIOCLOUD@1.1001:trchildren`
`FSCFOLIO@1.1001:EC_Root`
Configuration instances
Instances of classes with `COOTC@1.1001:AppCategory` base class
	Instance extension	Error

Configuration instances must not reference external class from non-friend component!	Configuration instance extension	Error
AppCategory instances may only assign the apps and templates properties with apps and classes from own or friend components!	AppCategory instance extension	Error
In the class chain of <instance> <class> could not be resolved. Add component <components> to project references! If this restriction fails, the other restriction evaluations on instances and instance extensions might return incorrect results.	Instance, Instance extension	Error

14.6.2 User Interface Language

Restriction	Restricted element	Severity

External ui element from non-friend component should not be extended!	Form/Property Page/Desk Form/ Menu/Menu Root/Task Pane extension Binding	Warning
Extension with address must not be used!	Form/Property Page/Desk Form/ Menu/Menu Root/Task Pane extension Binding	Error

14.6.3 Use Case Language

Restriction	Restricted element	Severity
Overriding external action from non-friend component of external classes from non-friend component is not allowed!	Overrides	Error
Wrapper must not be used!	Use Case Wrapper	Error
Extending external element from non-friend component is not allowed!	Dialog Extension	Error
Extension with address must not be used!	Dialog Extension	Error

14.6.4 Organization Structure Language

Restriction	Resticted element	Severity
Defining organizational structure model is not allowed!	Organizational Structure Model	Error

14.6.5 Business Process language

Restriction	Resticted element	Severity

Defining positions and/or organizational units for activities is not allowed!	Activity	Error
Extending external element from non-friend component is not allowed!	Activity Extension	Error
Extension with address must not be used!	Activity Extension	Error
The use of condition, case statement, loop and wait element is not allowed!	Process	Error

14.6.6 Customization Language

Restriction	Resticted element	Severity
No target add excepting for EditionFolioCloud@1.1 allowed!	Target Definition	Error
Overriding target is not allowed!	Target Definition	Error

14.7 Enterprise App Profile Restrictions

14.7.1 Object Model Language

Restriction	Restricted element	Severity

Only Compound-, Basic-, ContentObject own and friend classes can be used as base classes!	Class	Error
Extending external element from non-friend component is not allowed!	Class/Enumeration/Compound Type extension	Error
Extension with address must not be used!	Class/Enumeration/Compound Type extension	Error
Edition and/or Solution instance must not be used!	Instance	Error
Application must not be instantiated!	Instance	Error
Instances with address must not be used!	Instance	Error
Not component instance must not be used!	Instance	Error

Extending external instance from non-friend component is not allowed!
Exceptions:
`FSCFOLIOCLOUD@1.1001:trchildren`
`FSCFOLIO@1.1001:EC_Root`
Configuration instances
Instances of classes with `COOTC@1.1001:AppCategory` base class
	Instance extension	Error

Configuration instances must not reference external class from non-friend component!	Configuration instance extension	Error
AppCategory instances may only assign the apps and templates properties with apps and classes from own or friend components!	AppCategory instance extension	Error
In the class chain of <instance> <class> could not be resolved. Add component <components> to project references! If this restriction fails, the other restriction evaluations on instances and instance extensions might return incorrect results.	Instance, Instance extension	Error

14.7.2 User Interface Language

Restriction	Restricted element	Severity

External ui element from non-friend component should not be extended!	Form/Property Page/Desk Form/ Menu/Menu Root/Task Pane extension Binding	Warning
Extension with address must not be used!	Form/Property Page/Desk Form/ Menu/Menu Root/Task Pane extension Binding	Error

14.7.3 Use Case Language

Restriction	Restricted element	Severity
Overriding external action from non-friend component of external classes from non-friend component is not allowed!	Overrides	Error
Wrapper must not be used!	Use Case Wrapper	Error
Extending external element from non-friend component is not allowed!	Dialog Extension	Error
Extension with address must not be used!	Dialog Extension	Error

14.7.4 Business Process language

Restriction	Resticted element	Severity
Defining positions and/or organizational units for activities is not allowed!	Activity	Error
Extending external element from non-friend component is not allowed!	Activity Extension	Error
Extension with address must not be used!	Activity Extension	Error
The use of wait elements is not allowed!	Process	Error

14.7.5 Customization Language

Restriction	Resticted element	Severity

No target add excepting for EditionFolioCompliance@1.1 allowed!	Target Definition	Error
Overriding target is not allowed!	Target Definition	Error

15 Glossary

Access Control List (ACL)
Determines which user is granted which access rights for accessing an object.

Application Dispatcher
An instance of object class *Application Dispatcher* (`FSCVAPP@1.1001:ApplicationDispatcher`). The application dispatcher determines the looks of virtual applications and the available color schemes. The default application dispatcher is referenced in the *Virtual Application Default Configuration* (`FSCVAPP@1.1001:DefaultConfiguration`) referenced in the *Current Domain* (`COOSYSTEM@1.1:CurrentDomain`) object of your Fabasoft Folio Domain.

Branch
A button in a dialog of a virtual application.

Constraint
A rule for calculating or validating values, or for preventing invalid data entry into a property.

Dialog
An element of a virtual application for presenting a user interface to provide a means of communication between a user and a virtual application.

Fully Qualified Reference
A unique identifier for referring to a component object. The fully qualified reference consists of the software component prefix, followed by a colon and the reference of a component object, e.g. `COOSYSTEM@1.1:objname`.

Keyword
A reserved sequence of characters.

Reference
An identifier for referring to a component object.

Role
A combination of a position and a group or organizational unit. The combination of a position and an organizational unit is also referred to as abstract role as it only consists of abstract elements of an organizational structure.

Software Component Prefix
The part of a fully qualified reference that refers to the software component, e.g. `COOSYSTEM@1.1`.

Trigger
An action that is invoked when a predefined event occurs.

16 Bibliography

[ApAn09] The Apache Software Foundation: “Apache Ant User Manual”. URL: http://ant.apache.org/manual/index.html [Retrieved on December 17, 2009]

[AnLe10] Aniszczyk, Chris/Leszek, Pawel: “Debugging with the Eclipse Platform”. URL: http://www.ibm.com/developerworks/library/os-ecbug/index.html [Retrieved on January 18, 2010]

[Ecli10] Eclipse Foundation: “Eclipse Downloads”. URL: http://www.eclipse.org/downloads [Retrieved on January 18, 2010]

[Orac10a] Oracle: “Java SE Downloads”. URL: http://www.oracle.com/technetwork/java/javase/downloads/index.html [Retrieved on November 25, 2011]

[Orac10b] Oracle: “How to Write Doc Comments for the Javadoc Tool”. URL: http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#principles [Retrieved on November 25, 2011]

[Faba10a] Fabasoft: “Software Product Information: Fabasoft app.ducx”.

[Faba10b] Fabasoft: “White Paper: Fabasoft app.test“.

[Faba10c] Fabasoft: “White Paper: Fabasoft Folio Controls“.

[Faba10d] Fabasoft: “White Paper: Online Help in the Fabasoft Folio Web Client“.

[Faba10e] Fabasoft: “White Paper: Renamed, Deleted and Obsolete Component Objects “.